1998-12

# Design, implementation, and testing of MSHN's Resource Monitoring Library

Schnaidt, Matthew C. L.

Monterey, California. Naval Postgraduate School

# NAVAL POSTGRADUATE SCHOOL
# MONTEREY, CALIFORNIA


# THESIS

**DESIGN, IMPLEMENTATION, AND TESTING OF MSHN's RESOURCE MONITORING LIBRARY**

by

Matthew C. L. Schnaidt

December 1998

Thesis Advisor: Debra Hensgen
Thesis Co-Advisor: John Falby

# REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188



| 1. AGENCY USE ONLY *(Leave blank)* | 2. REPORT DATE December 1998 | 3. REPORT TYPE AND DATES COVERED Master's Thesis |
|---|---|---|

4. TITLE AND SUBTITLE: DESIGN, IMPLEMENTATION, AND TESTING OF MSHN's RESOURCE MONITORING LIBRARY

5. FUNDING NUMBERS

6. AUTHOR(S) Schnaidt, Matthew C. L.

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Naval Postgraduate School
Monterey, CA 93943-5000

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION/AVAILABILITY STATEMENT:
Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE:

13. ABSTRACT *(maximum 200 words)*

The Management System for Heterogeneous Networks (MSHN) requires the gathering of resource usage information of applications that run within the MSHN system and status information of the resources within the scope of the MSHN scheduler. The MSHN scheduler uses this information to make decisions. This thesis investigates one method of gathering the required information: a client library.

This research develops the mechanism and policy for the client library's resource monitoring role and carefully documents how applications can be easily linked with this client library. During run time the client library gathers information on an application's resource utilization by intercepting system calls and through the use of operating system functions. Resource information gathered includes total runtime, local and remote disk use, network use, memory use, CPU use, and time blocked waiting on user input.

The client library also determines end-to-end perceived status of the resources that the application uses. Specifically, this thesis develops a policy for passively gathering network performance characteristics, i.e., latency and throughput.
The per system call overhead added varied from less than 1% to 326%, with an average of 3% overhead added to the run-time of test programs.

14. SUBJECT TERMS wrapper, passive monitoring, intercept system calls, library, resource monitoring, MSHN, heterogeneous computing, resource management system

15. NUMBER OF PAGES 132

16. PRICE CODE

| 17. SECURITY CLASSIFICATION OF REPORT | 18. SECURITY CLASSIFICATION OF THIS PAGE | 19. SECURITY CLASSIFICATION OF ABSTRACT | 20. LIMITATION OF ABSTRACT |
|---|---|---|---|
| Unclassified | Unclassified | Unclassified | UL |

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18 298-102

**Approved for public release; distribution is unlimited.**

# DESIGN, IMPLEMENTATION, AND TESTING OF MSHN's RESOURCE MONITORING LIBRARY

Matthew C. L. Schnaidt
Captain, United States Army
B.S., United States Military Academy, 1988

Submitted in partial fulfillment of the
requirements for the degree of

**MASTER OF SCIENCE IN COMPUTER SCIENCE**

from the

**NAVAL POSTGRADUATE SCHOOL**
**December 1998**

## ABSTRACT

The Management System for Heterogeneous Networks (MSHN) requires the gathering of resource usage information of applications that run within the MSHN system and status information of the resources within the scope of the MSHN scheduler. The MSHN scheduler uses this information to make decisions. This thesis investigates one method of gathering the required information: a client library.

This research develops the mechanism and policy for the client library's resource monitoring role and carefully documents how applications can be easily linked with this client library. During run time the client library gathers information on an application's resource utilization by intercepting system calls and through the use of operating system functions. Resource information gathered includes total runtime, local and remote disk use, network use, memory use, CPU use, and time blocked waiting on user input.

The client library also determines end-to-end perceived status of the resources that the application uses. Specifically, this thesis develops a policy for passively gathering network performance characteristics, i.e., latency and throughput.

The per system call overhead added varied from less than 1% to 326%, with an average of 3% overhead added to the run-time of test programs.

## TABLE OF CONTENTS

## LIST OF FIGURES

## LIST OF TABLES

## ACKNOWLEDGEMENTS

First, I thank my supportive family: Anita, Dominick and Isabel thanks for your patience. Thank you to Dr. Debra Hensgen and John Falby for making computer science both interesting and fun. Finally, thanks to the entire MSHN team. The sense of camaraderie and teamwork gave this research meaning and made the process enjoyable.

# I. INTRODUCTION

This thesis investigates the problem of gathering end-to-end Quality of Service (QoS) resource usage information about an application program as it executes. Further, it presents a methodology to solve this problem and integrates the solution as part of the Management System for Heterogeneous Networks' (MSHN) proposed architecture.

## A. BACKGROUND

In October 1990, the Air Force Deputy Executive Officer, Major General May, sent a team of operator-analysts to Riyadh, Saudi Arabia in support of the upcoming Desert Storm air campaign. Once in Riyadh, this team co-located with the team of joint military planners charged with developing the Air Tasking Order (ATO). The ATO is the comprehensive plan for all friendly air assets; these assets include fighter, attack, support, fuel, intelligence, command and control, and reconnaissance aircraft. The operator-analysts concentrated on populating a database of a modeling and simulation package called the Command, Control, Communications, and Intelligence Simulation (C3ISIM), which later became known as Extended Air Defense Simulation (EADSIM). In early November, the EADSIM team conducted simulations of the air war's first three hours. These simulations produced data supporting the suspicion of heavy friendly losses in certain areas. The joint military planners viewed EADSIM as a credible mission-planning tool and used these simulation results in their ATO planning process. [CASE91]

Following the commencement of Desert Storm, EADSIM lost much of its usefulness as a mission-planning tool due to the high pace of operations, a rapidly changing ATO, and the large amount of time required for each simulation. The team found that it took eight hours on the deployed computers to simulate three hours of real-time air operations [PORT97]. EADSIM, which had proved to be such a useful tool in planning the first day of the air war, became unusable in the later days for its originally intended purpose because it simply took too long to run the simulation on the deployed assets.

The Department of Defense has a great number and variety of computing hardware. Some of this hardware was designed to run a particular set of application programs, while other hardware is general purpose in nature and may run a broad variety of applications. These machines are distributed over a variety of physical locations, some grouped together in

computing enclaves, while others sit alone at independent sites. When a network connects many of these machines, it creates a distributed, heterogeneous computing base. To make the most efficient use of this base and provide the best performance to each user, application programs must be matched with the right hardware in the right location. In this way, the users can obtain quicker results and higher fidelity answers without investing in new hardware. With decreasing defense budgets and a vast, connected, heterogeneous hardware base, a system that meets this challenge will provide lower cost information superiority in the future. Such a system will enable applications such as EADSIM to prove even more useful.

A system that more wisely uses resources should provide users with a transparent interface. Users would request the system to execute an application, possibly specifying a deadline and a preferred version (i.e., full video vs. text). The system would then use available resource requirement information to determine, based on past performance and perceived current load, the computing resources most suitable for running the application. It would also support execution of the application and, if possible, return the results within the given deadline. The proposed Management System for Heterogeneous Networks (MSHN) addresses this requirement.

## B. MSHN'S OVERALL GOAL

The goal of MSHN is to provide a computing environment that delivers each user's specified quality of service, subject to:

- resource availability,
- the user's individual priority, and
- the preference of each user for different forms of the requested results. [SCHN99]

Given a set of disparate jobs and a set of shared, heterogeneous resources, MSHN will determine where and when to run each job in a way that maximizes a metric, which incorporates a collection of application-specific quality of service measures [KRES98]. MSHN will consider end-user priorities, adaptive applications, preferences and deadlines. Integral to MSHN is the requirement that its scheduler knows which resources an application needs to run and the current load on those types of resources that are available to MSHN.

## C. SCOPE OF THIS THESIS

MSHN requires the gathering of resource usage information for applications that run within the MSHN system as well as resource status information within the scope of the MSHN scheduler. The MSHN scheduler uses this information to make scheduling decisions. The methods used to implement the gathering of this information are subject to three constraints:

1. The implementation must not require any changes to the operating system.
2. Modifications to the application code must be minimized.
3. The overhead imposed by the information gathering mechanism should not be excessive.

There are several reasons for the first requirement:

- The ultimate goal is the widespread acceptance of this system. Many potential users are reluctant to allow modifications to their operating system.
- When routine operating system upgrades do occur, we do not want to have to redesign and redistribute our system to include the features or improvements of this new release.
- We do not want to risk compromising the security features of the operating system by changing the kernel.
- Source code of all operating system releases may not be available for modification.

The second and third requirements address acceptance and usability issues. If the application writer must modify his code, or if the use of our system incurs unacceptable overhead, the system will not be used.

Given these requirements and constraints, the approach for this research is to use a client library that intercepts system calls [SILB98]. The client library is linked with the object code of the application that is to be monitored prior to run-time. During run-time the client library gathers information on an application's resource utilization by intercepting system calls and through the use of operating system functions. Additionally, the client library determines end-to-end perceived status of the resources that the application uses.

The use of a library that is linked with the application by the linker meets the first two constraints, as no change to the operating system or source code is required when using this

technique. Also, because we are not requiring separate processes, our added overhead should be relatively low. Chapter VII discusses observed overhead.

## D. MAJOR CONTRIBUTIONS OF THIS THESIS

There are three major contributions of this thesis. It develops a new technique to passively monitor the end-to-end network performance (throughput and latency). Our method to monitor network performance includes some of the best features of current tools and application components. Secondly, it describes the implementation of a client library that monitors an application's resource usage as well as the load on the resources on which the application runs. Finally, it provides extensive documentation on how to transparently wrap (or intercept) system calls. The Condor System, discussed in Chapter III, was a primary reference for determining this method.

## E. ORGANIZATION

This thesis is organized as follows: Chapter II describes background and puts this research into perspective. It also overviews the proposed MSHN architecture. Chapter III discusses related work. It introduces several resource management systems and compares them to the MSHN architecture. Chapter IV describes how to wrap (or intercept) system calls. Chapter V details the monitoring of an application's resource usage including what resources the library monitors, and how the library uses wrapped system calls and operating system functions. It also discusses the architecture of the client library, and the roles of the different components of the architecture. Chapter VI describes methods used to determine the end-to-end perceived status of the resources. This chapter also introduces the technique we developed to passively monitor end-to-end network performance. We discuss our experimental methods for determining the overhead added by our library, as well as the results of those experiments in Chapter VII. In the final chapter, we summarize our results and propose future work.

# II. MANAGEMENT SYSTEM FOR HETEROGENEOUS NETWORKS

## A. PURPOSE

Modern computer networks have grown in size and complexity as fast as the technology will allow. They span great distances and include machines of varying types. This expansion has resulted in the need to effectively manage large heterogeneous networks of computers in order to deliver good performance to all users, regardless of their individual Quality of Service requirements. In response to this need, a team of computer scientists, funded under Defense Advanced Research Projects Agency's (DARPA) QUORUM program, is currently developing a resource management system named MSHN[1].

The goal of MSHN is to provide a computing environment that delivers each user's specified quality of service, subject to the available resources, the user's individual priorities, and the preference of each user for different forms of the requested data. Given a set of jobs, MSHN will determine where and when to run each job along with the appropriate version of the job to run. MSHN evolved from SmartNet, which was a heterogeneous framework for minimizing the time at which the last job of a set of computationally intensive jobs finishes on a suite of heterogeneous computing resources [KIDD96]. SmartNet treats the set of compute resources available as one virtual heterogeneous machine (VHM). SmartNet achieves superior performance by mapping applications to resources based upon knowledge of the VHM and job characteristics. MSHN differs from SmartNet in several ways: (1) it strives to support Input/Output intensive and real-time jobs, in addition to compute-intensive jobs; (2) it accounts for the fact that a job may need many different resources, not just a central processing unit, to execute; and (3) it manages *adaptive* applications.

One of the important improvements of MSHN over traditional resource management systems is that MSHN will support adaptive applications. Adaptive applications are those that can produce results using one of a variety of algorithms or in one of a variety of forms. MSHN uses knowledge of these various forms to choose the appropriate one for the given operating environment. MSHN is intended to help achieve the goals set forth in Joint Vision 2010

[1] Pronounced "mission."

[JOIN95], C4I For The Warrior [JOIN97], and DD-21 [BUSH97], by allowing a commander to maximize the utility of his computer network. This is particularly important when the commander is faced with a volatile wartime environment.

In addition to improving the performance of other applications, MSHN will expand the usefulness of compute-intensive, batch processed jobs such as C3ISIM. Had MSHN been available during the Gulf War, it may have been possible to use C3ISIM's simulation results throughout the air campaign, as opposed to only prior to the start of the battle.

## B. MSHN'S PROPOSED ARCHITECTURE

The MSHN architecture is still under development. The description that follows is based upon the architecture design as of September 5, 1998 [SCHN99]. While the ultimate MSHN architecture may differ slightly from this description, the major components and concepts are expected to remain unchanged.

MSHN will consist of a client-server architecture. It will be composed of at least the following components:

- a Client Library,
- a Scheduling Advisor,
- a Resource Requirements Database,
- a Resource Status Server, and
- a MSHN Daemon.

The following paragraphs provide an abstract description of each of the components, and Figure 1 provides an overview of the entire architecture. Although these components are shown together, they may in fact reside on separate machines. Usually, many different client applications will be running at any given time. Additionally, it is conceivable that some of the components may be replicated.

### 1. Client Library

The client library is linked with both adaptive and non-adaptive applications. It provides a transparent interface to all of the MSHN services [KRES97]. The client library performs at least the following functions: (1) it intercepts system calls to record resource requirements; (2) it

forwards requests to start another process, when appropriate, to the Scheduling Advisor; and (3) it intercepts and performs the appropriate action on requests from the Scheduling Advisor to adapt. It forwards the recorded resource requirements to the Resource Requirements Database. The final implementation of MSHN will be able to forward the performance measurements and resource requirements through a proxy when that is more efficient.

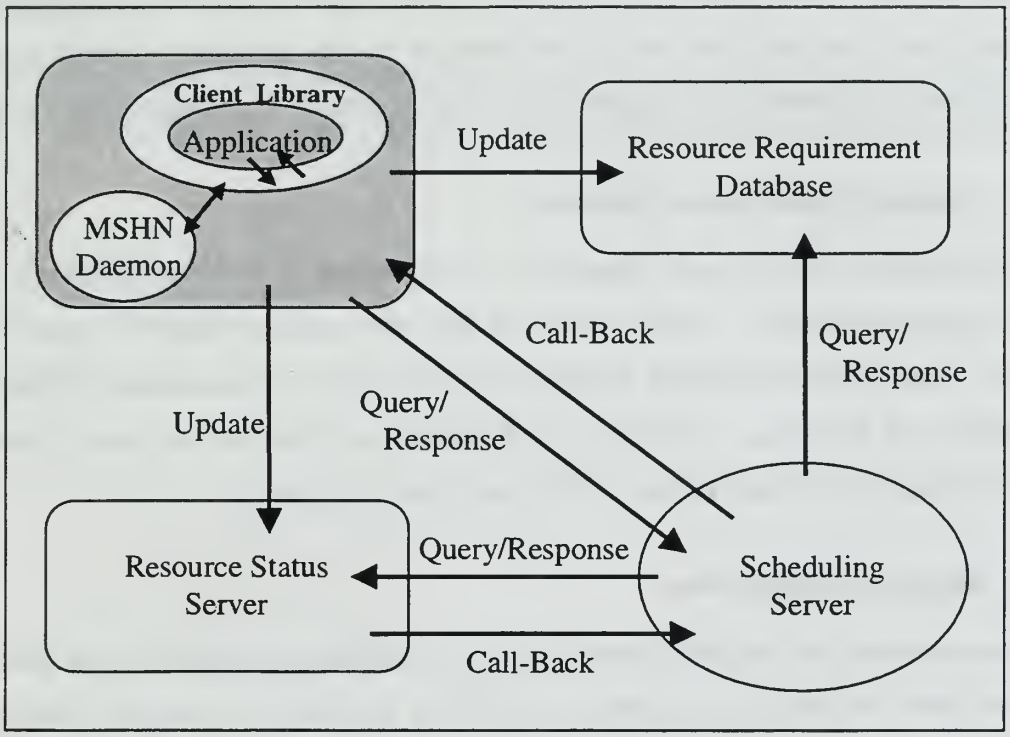


**Figure 1: MSHN Architecture**

## 2. Scheduling Advisor

The Scheduling Advisor performs the highly complex task of scheduling multiple jobs, from multiple users, onto one (or several) computers from a pool of heterogeneous computing platforms. The sophisticated algorithms that the Scheduling Advisor will use to make decisions are beyond the scope of this thesis. However, this research requires knowledge of the interfaces presented by the Scheduling Advisor.

The Scheduling Advisor will accept scheduling requests from the client libraries. The Scheduling Advisor will query both the Resource Status Server and the Resource Requirements Database. These queries must respond with near real-time information on the status (load) of the VHM, and the resource requirements of the application. Once the Scheduling Advisor receives

this load information, it can calculate a mixture of computing and network resources that will, with high probability, deliver the requested quality of service.

Additionally, in case of a significant deviation from the initial resource status estimate, the Scheduling Advisor will receive notification from the Resource Status Server. For example, if a communications path is severed, or a machine fails, the Scheduling Advisor will be notified and can recalculate a new scheduling solution for the affected applications. The Scheduling Advisor may then signal the client library and advise it that the application should begin using a different algorithm, or perhaps recommend that it shift execution to a different set of resources.

### 3. Resource Requirements Database

The Resource Requirements Database is a repository of information pertaining to the execution of user applications. A job consists of the code and data required to execute a user's application. This database contains statistics on the run-time characteristics of jobs, such as CPU, memory, and disk usage. The Resource Requirements Database provides this information to the Scheduling Advisor upon request. The client libraries update it.

### 4. Resource Status Server

The purpose of the Resource Status Server is to maintain a repository of the three types of information about the resources available for MSHN to schedule: the relatively static (e.g. CPU speed), the moderately dynamic (e.g., operating system version), and the highly dynamic information (e.g., network load). The Scheduling Advisor will query the Resource Status Server to obtain an initial estimate of the currently available computing and networking resources. After making a scheduling decision, the Scheduling Advisor will notify the Resource Status Server of the additional loads that it expects the client application to place on the compute and networking resources. Much of this thesis is dedicated to determining the best mechanisms for obtaining this most dynamically changing type of information for network resources. [KRES97]

Periodically during the execution of an application, the client library will update the Resource Status Server with status of the computing and networking resources in use by the application. Also, as described in subsection 2 above, the Resource Status Server sets up a callback with the Scheduling Advisor. If the perceived loads on the resources fall outside a specified range, the Resource Status Server will notify the Scheduling Advisor.

### 5. The MSHN Daemon

The MSHN Daemon executes on all compute resources available for scheduling by the MSHN Scheduling Advisor. It is used to begin and control the execution of processes that are submitted to MSHN.

## C. SUMMARY

MSHN is a resource management system currently under development. Its purpose is to manage (adaptive) jobs in a heterogeneous environment, so that the system delivers good quality of service.

# III. RELATED WORK

## A. DISTRIBUTED RESOURCE MANAGEMENT SYSTEMS

The motivation for building a distributed resource management system (RMS) is to transparently improve availability, reliability, fault tolerance and scalability of a system of interconnected computing resources, without modifying the operating system. MSHN is an RMS. This chapter will review several existing RMSs that influenced the design of MSHN. There will always be some run-time overhead in gaining the above named benefits from an RMS. Therefore, in this chapter, we also review some research that we use to minimize the run-time overhead added by MSHN.

## B. SMARTNET

MSHN evolved directly from lessons learned designing and implementing SmartNet, though MSHN's goals generalize those of SmartNet.

### 1. Overview of SmartNet

SmartNet is a scheduling framework for heterogeneous systems developed by the Heterogeneous Computing team at the US Navy's facility at the Naval Command, Control, and Ocean Surveillance Center (NCCOSC) for Research, Development, Testing and Evaluation (RDT&E) in San Diego [NAVA96]. SmartNet's goals are to maximize computing power and increase the throughput of a set of jobs by better leveraging existing resources [ARMS97].

When manually submitting jobs to one (or several) machines in a network, a user often considers two points:

- the current loads on available machines, and
- their understanding of their program's performance on the different platforms.

SmartNet was the first heterogeneous resource management system to include both of these considerations in its scheduling methodology [KIDD96].

SmartNet interfaces with both users and resources (Figure 2). The user interfaces include a system administrator interface as well as a job-submitting user interface. The system

administrator interface allows someone to administer the entire system. The job-submitting user interface allows a user to specify a requested priority, but the system administrator may override that request.

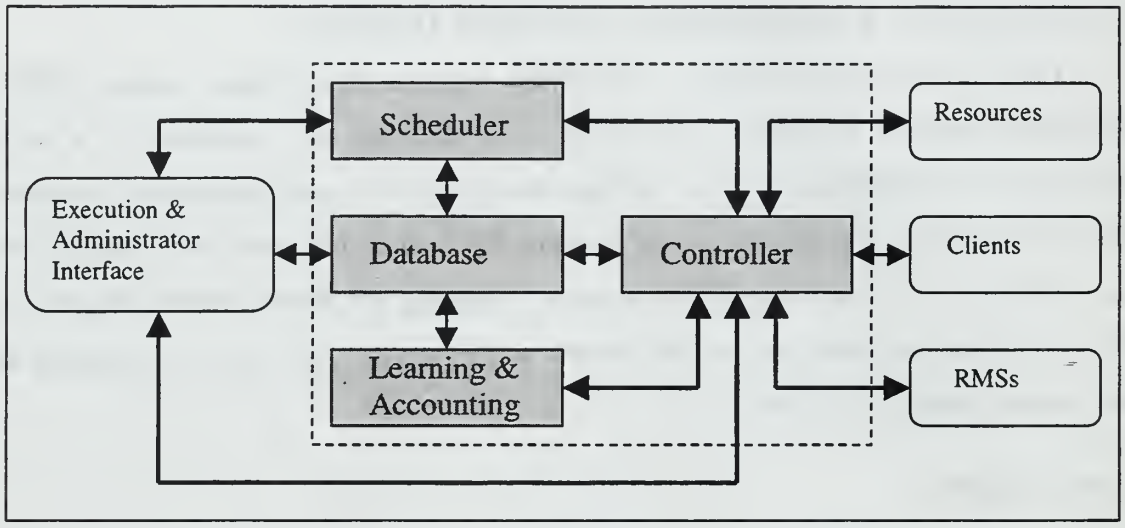


**Figure 2: SmartNet Architecture, modified from [KIDD96]**

SmartNet consists of four processes:

1. The SmartNet Controller process interfaces with resources. The resources that the controller process manages include physical ones such as machines and virtual ones such as RMS's (including additional instantiations of SmartNet).

2. The SmartNet Scheduler contains both optimization and scheduling algorithms. The scheduler includes Exhaustive, Greedy, Evolutionary, and Simulated Annealing algorithms. The Scheduler is designed so that new algorithms may be easily integrated as they become available or necessary.

3. The SmartNet Database is an ASCII text file containing information about sites, groups, machines, jobs, and model-machine pairs. SmartNet tracks expected time for completion (ETC) data in the model-machine listings that the Scheduler uses to create near optimal job-machine assignments.

4. The SmartNet Learning and Accounting process tracks and reports rogue processes. A rogue process is one that exceeds its ETC by more than a certain tolerance and endangers the schedule developed by the Scheduler. Additionally, the Learning and Accounting process gathers experiential data on the actual run-time of jobs. It uses this data to update the ETC field of the model-machine listings in the Database.

### 2. Observations

SmartNet shares many similarities with the objectives of MSHN. The most significant similarities are:

- SmartNet and MSHN both exploit heterogeneity of resources and jobs to improve overall system performance.
- SmartNet and MSHN both use experiential data to improve future scheduling.
- SmartNet and MSHN both provide flexibility in their respective scheduling processes. This allows for future improvements and handling of unforeseen conditions.

SmartNet differs from MSHN in a number of significant ways:

- MSHN supports adaptive applications.
- SmartNet tracks a single large-grained resource usage measurement, ETC. MSHN experiments with multiple fine-grained resource usage measurements.
- MSHN can apply experiential data learned for one machine to other machines.

## C. ODYSSEY

### 1. Overview of Odyssey

A team from Carnegie Mellon University developed Odyssey as a prototype implementation of an application-aware adaptation system for mobile computing [NOBL97]. The developers state that adaptation is the key to mobile computing. Acceptable service can only be provided by a system that is aware of changes in resources and can quickly adapt to meet those changes. Examples of these changes include unpredictable variation in network quality and bandwidth, and disparity in availability of remote services. System weight and size limitations, limited battery power, and environmental exposure are examples of other challenges

imposed on mobile computing. The Odyssey team believes that a system of servers[2] that is capable of delivering domain-specific resource-aware adaptation can meet these challenges and provide the best service to users of mobile computers.

In addition to the Odyssey system, the team developed and implemented several adaptive mobile information access applications to test Odyssey. The Odyssey team demonstrated a video player, web browser and speech recognizer. The adaptive applications change their behavior based on changes in resource availability. The applications adapt automatically; the user may be aware of the adaptation, but does not initiate or control it.

Odyssey developers define some important notions.

- **Agility** is the essential attribute of adaptive systems and is defined as the speed and accuracy with which a system detects and responds to changes in resource availability.

- **Fidelity** is the degree to which data presented matches the reference copy at the server.

- The definition of **performance** depends upon the application executed. A speech-recognition application may define performance in terms of the percentage of words recognized, while a streaming video application may define it in terms of the number of frames per second received by the user.

Odyssey was implemented on the NetBSD kernel. The kernel was modified to include an interceptor module and several new system calls to manipulate Odyssey objects. The bulk of Odyssey software resides outside the kernel and consists of a Viceroy and Wardens (Figure 3). The Wardens are statically linked with the Viceroy. The Wardens communicate with the servers; an application never directly accesses a server. The Viceroy has a global view of resources available to the processes running on the client machine; it is the Viceroy's job to manage these resources and to advise the client applications to adapt when required.

Odyssey incorporates a form of type-awareness that differentiates between different uses of data. The authors justify this use of types with an example of network traffic. Video data can be shipped using a streaming protocol that simply drops very late messages rather than re-transmitting them. The Odyssey team notes that this method, however, would not work for database or file updates. The Wardens manage data types associated with their server.

---

[2] In this context, servers are the remote programs that provide data or do processing for the mobile user's applications.

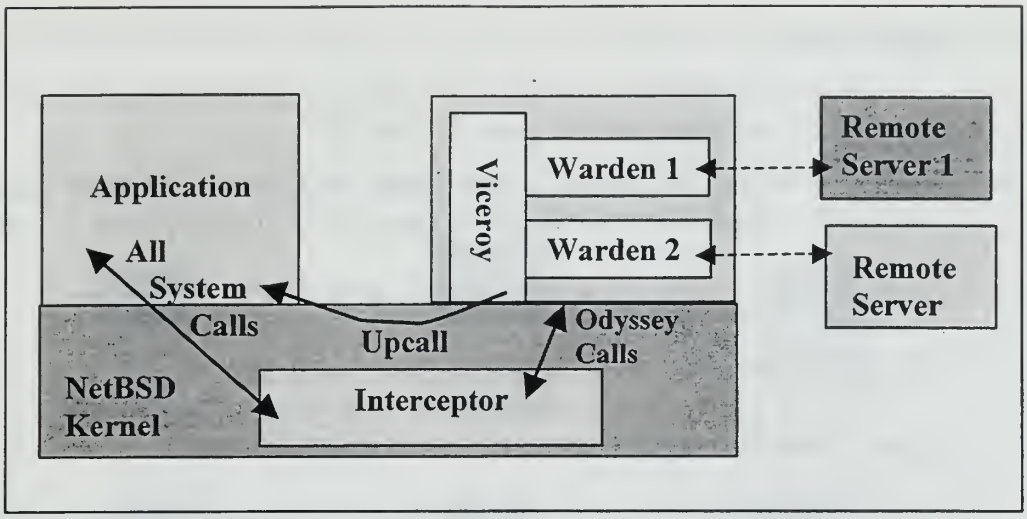


**Figure 3: Odyssey Architecture [NOBL97]**

## 2. How Odyssey Works

We consider an adaptive application that runs on the Odyssey system. The application issues a request for a streaming color video feed, along with an estimate of network bandwidth required to support the request. This request is caught by the interceptor, and is passed out of the NetBSD kernel to the Viceroy. The Viceroy checks the availability of network bandwidth and compares it to the estimate in the application's request. If the Viceroy cannot support the bandwidth requirements of the streaming video, it sends an error message back to the adaptive application indicating the amount of bandwidth available. The adaptive application may then choose to adapt to a lower performance level that is within the bandwidth returned in the error message. For example, the application could respond with a request for black and white video. If the Viceroy estimates that it can support this bandwidth requirement, it registers this request and passes the request to the appropriate Warden. For example, the Warden communicates with the server to service the request.

The Viceroy constantly monitors system performance and application resource usage. If the Viceroy notices a drop in resource availability, it will issue a call to its application advising it to adapt. The application would then submit a request for lower performance service; in the case of our example, perhaps still photographs. If the Viceroy later notices an improvement in resource availability, it will issue calls to the applications, again advising them to adapt.

### 3. Observations

The Odyssey system shares several goals with MSHN:

- Odyssey addresses the need to monitor more than just one resource metric: network bandwidth, network latency, disk cache space, CPU, battery power, and money.
- The functionality of the wardens is a subset of MSHN's Client Library.

The differences between Odyssey and MSHN are listed below:

- Odyssey requires modifications to the operating system kernel.
- The Viceroy instantaneously samples resources and bases adaptation decisions on that sample. MSHN will use filtering theory, thereby incorporating multiple samples.
- The Odyssey clients all run at the same priority level.
- Odyssey does not support real-time applications.
- Making decisions for each application independently can lead to system instability. MSHN will simultaneously consider the entire set of applications that use the same resources.

## D. CONDOR

### 1. Overview of Condor

Condor is a distributed batch processing system developed at the University of Wisconsin-Madison that makes use of idle UNIX workstations. It is a software system that runs jobs on a cluster of workstations to make use of otherwise wasted CPU cycles. Condor executes jobs that are submitted via one workstation on a pool of workstations (Figure 4). One workstation acts as the Central Manager; it watches over all other workstations in the pool. It decides when and where to run submitted jobs. Each member of the pool may be characterized as a busy machine, an idle machine, a submitting machine or a remote execution machine. Idle machines are machines that currently have no interactive users, have CPU usage below a predetermined threshold, and are not running any Condor jobs. A busy machine is one that currently has a user at the keyboard or whose CPU usage is above the predetermined threshold. A submitting machine is the machine from which a user submits a job to the Condor Central

Manager. A remote execution machine is a machine that was idle, but is now executing a job given to it by the Central Manager. [LITZ92] [LITZ97] [LIVN95] [PRUY96]

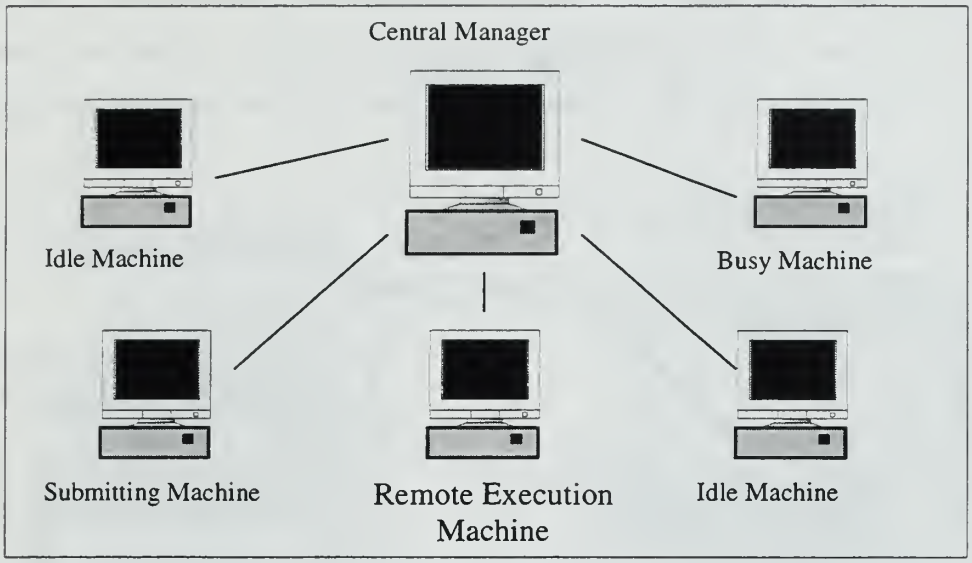


**Figure 4: The Condor Pool**

One of the objectives of Condor is to minimize any penalty to the workstation owner who allows his or her workstation to become part of the pool. For this reason, a job must be able to be moved immediately to another workstation if its current workstation becomes busy. To avoid losing forward progress made on a job, the remote execution machine writes a checkpoint file to disk before killing the job[3]. The checkpoint file has all of the job's state information that is required to restart the job on another machine. Each process that runs through Condor must be linked with a Condor library that executes remote procedure calls and checkpointing. All of the workstations in the Condor pool run several control daemons (UNIX processes that run in the background) in order to monitor workstation activity. These daemons determine when a machine becomes busy or idle.

## 2. How Condor Works

A user submits a job to the Condor pool through the Condor Central Manager. The user can specify preferences to the Central Manager, for instance, the amount of memory or disk

[3] Condor can be configured to checkpoint programs periodically instead of, or in addition to, when a job is terminated due to its current workstation becoming busy.

space desired for efficient execution. However, these can only be suggestions; the job must be capable of running on any member of the Condor pool. The Central Manager queues all jobs to run in the pool and then periodically allocates those jobs to idle machines. If no machines are idle, the jobs remain queued until machines become available. When the Central Manager allocates a user's job to a machine, it considers that platform to be the remote execution machine (Figure 5).

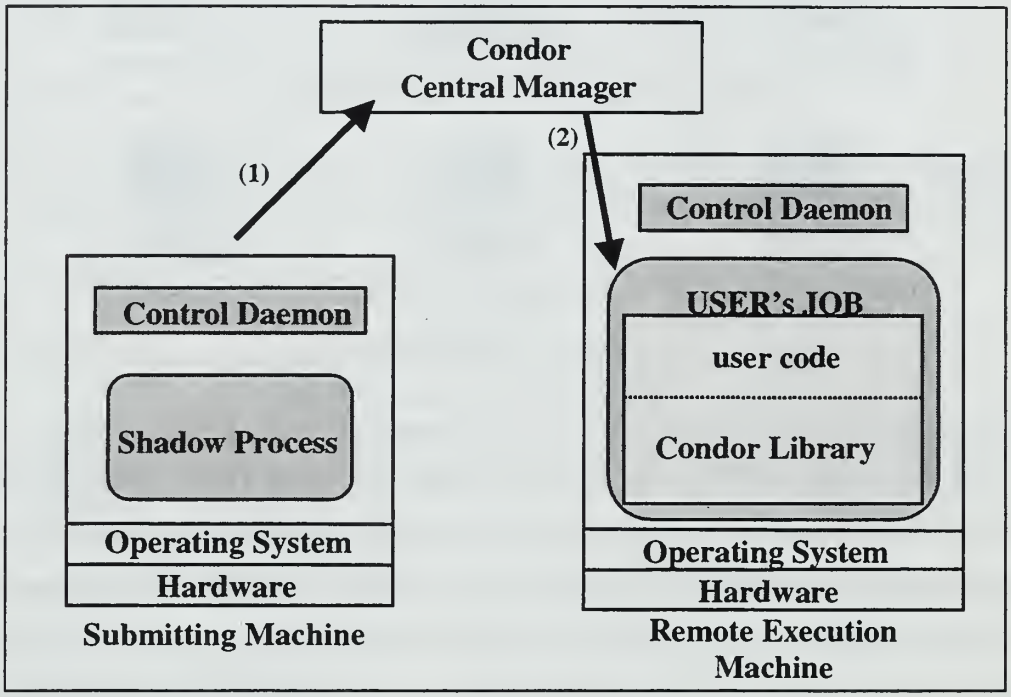


**Figure 5: Running a Job on Condor [LIVN95]**

(1) Job is submitted to Condor Central Manager, (2) Central Manager locates idle machine, starts users job, records remote execution machine.

Condor must support environments that do not have a completely consistent network file system. Additionally, people submitting jobs may have no permissions to access the local file system of the remote execution machine, so input and output files may not be directly accessed on that machine. Also, if a job needs to prompt the user for information to be entered via the keyboard, those messages must show up on the submitting machine's console rather than on the remote execution machine. This problem is solved using remote procedure calls. The Condor library is statically linked with the user's job and "stubs out" most system calls. The Condor library uses these stubs to "catch" system calls, and prevent them from being executed on the remote execution machine. The stubs use remote procedure calls to a "shadow process" running

on the submitting machine. The submitting machine executes the system calls and actually does all input and output (Figure 6). [LITZ97]

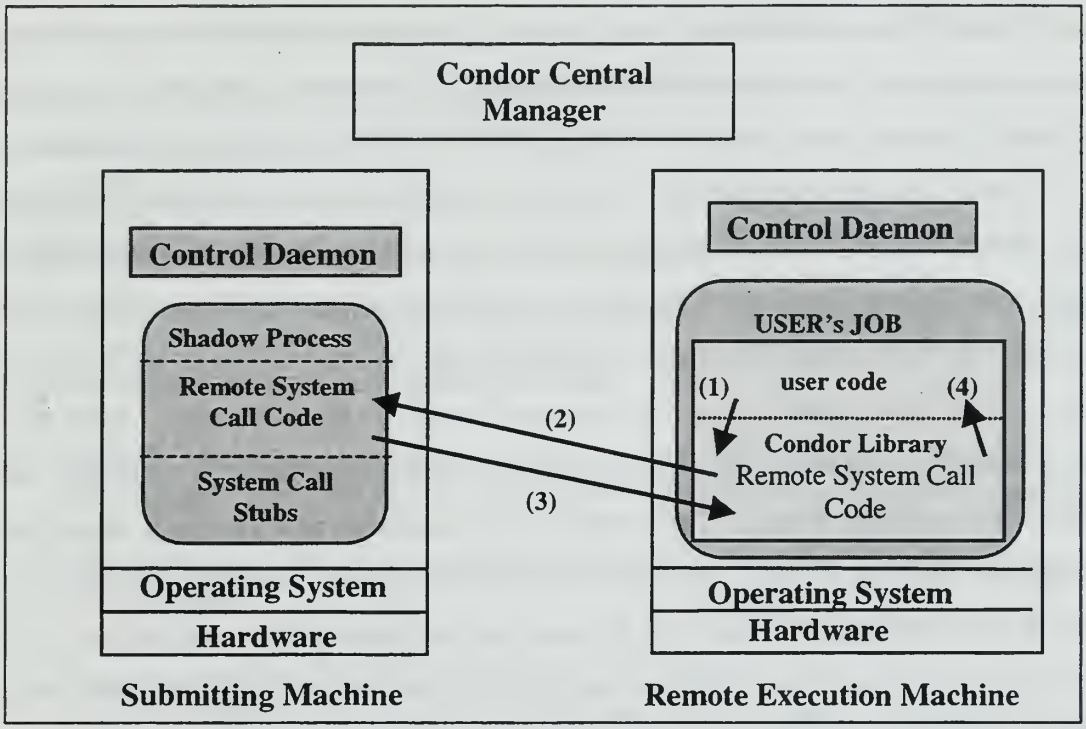


**Figure 6: Condor Handles Distributed File Ownership [PRUY95]**

(1) User's job makes an I/O system call. (2) The Condor library stub sends request back to Shadow Process. Shadow Process executes the system call on the Submitting Machine. (3) Shadow Process returns the result of the system call. (4) Condor library returns result of system call to the user's code.

Condor controlling daemons continuously monitor system activity. When a remote execution workstation becomes busy, the daemon sends a signal to the Condor library portion of the process. This signal informs the library to write checkpoint information to the checkpoint file on disk, and to terminate the process. The checkpoint file contains the state information that Condor will need to restart the process. The checkpoint file is then transferred back, as a file, to the machine from which the Condor job was originally submitted.

When the Condor Central Manager locates a newly available idle machine, the job's original binary executable as well as the most recent checkpoint file for that job, if any, is transferred to that machine. The Condor library knows whether it is the first time the process with which it is linked has run. If it is not the first time, the library will perform a restart by reading in the checkpoint file and resetting the process to the state it was in prior to checkpointing.

### 3. Wrapping System Calls

In order to gather the state data that the Condor library will need to create a checkpoint file and to convey environmentally sensitive (e.g. file operations) information to the shadow process, Condor uses a technique called "wrapping." Wrapping captures information from system calls. We will now discuss the mechanisms involved in wrapping system calls.

System calls are wrapped by writing a function that has the same signature (same function name, number of parameters, and return type) as the system call. The wrapper function is linked with the user's code prior to run-time. Thus, any system calls made by the application are "caught" by this wrapper function; the function then has full access to all of the parameters passed in to the system call, as well as the return value of the system call. After the wrapper function has done whatever work it will do with the input parameters, it must pass the system call on to the operating system. The wrapper function cannot make a direct system call as the user code did; to do so would be recursive and would cause the wrapper function to call itself. Instead, in the UNIX environment, the wrapper function must make a call to the `syscall()` routine or to a C library that has been modified to change the original system call's name. Chapter IV will discuss intercepting system calls in detail.

### 4. Observations

Although the goals of Condor are very different from those of MSHN, many of the mechanisms developed for Condor prove quite useful in this research. These include:

- Condor's use of system call wrapping, and
- Condor's static linking of jobs with libraries.

## E. SYNTHETIX

Unlike the previously mentioned systems and MSHN, Synthetix is not an RMS. However, we describe it in this chapter because we used ideas it developed in order to minimize the overhead of our client libraries. We also expect that follow-on research, to that described in this thesis, may directly use the Synthetix tools.

## 1. Overview of Synthetix

Synthetix is an approach used to specialize an operating system. It requires modifications to the operating system prior to run-time. At run-time, system calls are automatically modified based upon changing conditions, known as quasi-invariants, to make use of more efficient code; this modification results in **specialization**. Associated with this approach are some tools that the Synthetix team developed to assist in implementation.

Before proceeding further, we must define some terms.

- **Quasi-invariants** are system states and variables that are anticipated to be true for a period, but which may become invalidated. Quasi-invariants are used to define when specialized modules can correctly be used.

- **Guards** check quasi-invariants. They are placed everywhere that quasi-invariants can change. When a guard detects a quasi-invariant change, it triggers replugging.

- **Specialization** refers to code that is written to perform correctly in a restricted environment.

- **Incremental Specialization** refers to the replacing of a module with a more specialized module as guards discover true quasi-invariants. Because a specialized module that depends on quasi-invariants can be removed even before it is used, the Synthetix team refers to the use of such specialized modules as **optimistic specialization.**

- **Replugging** is the process of replacing one procedure with another at run-time; replugging is where incremental specialization happens [PUCA96B].

The Synthetix approach allows the operating system to be altered dynamically during run-time to improve efficiency; the alteration is in the form of specialized replugged modules. A system call results in guards checking the related quasi-invariants that indicate whether to use the currently plugged module, to incrementally specialize further, or to revert to the system's routine. An example follows that will clarify Synthetix's approach. [PUCA96A] [PUCA96B]

## 2. How Synthetix Works

### *a. Prior to Run-time*

The operating system is modified by adding a module that allows system calls to be routed to modules outside the kernel. The system calls that will be specialized are written as module templates. Quasi-invariants are defined and associated with ranges of values for which these templates are valid. Guards are placed in the code wherever a quasi-invariant may change (Figure 7).

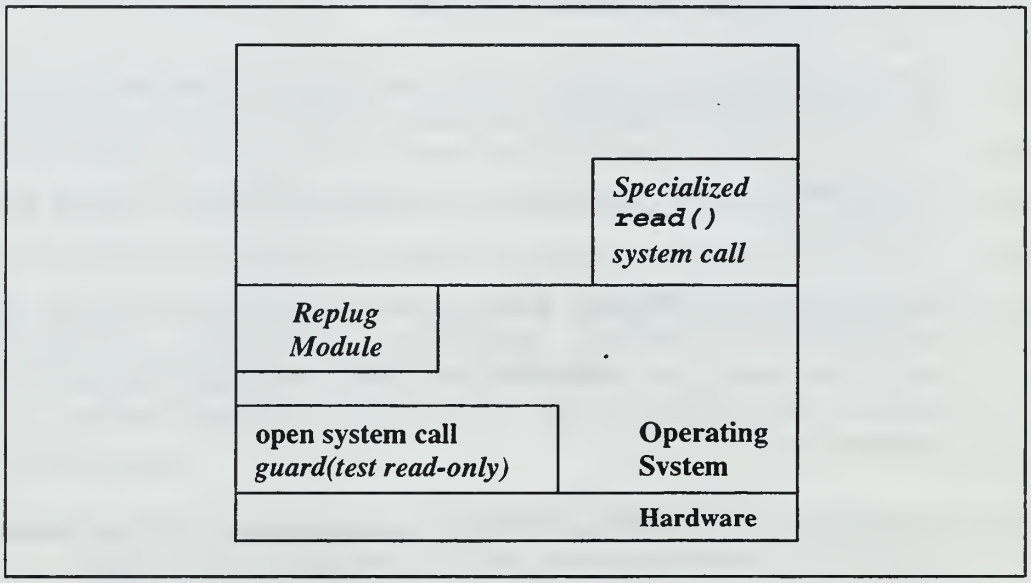


**Figure 7: Additions Prior to Run-time**

(Italics indicate additions made prior to run-time)

We now paraphrase an example given by the Synthetix team. They specialize the `read()` system call for a read-only file. The quasi-invariant is "File is read-only." The `open` system call is the only place this quasi-invariant can be changed. Code is inserted into the `open()` system call to test whether a file is being opened read-only; this code is the quasi-invariant's guard. In addition, a specialized version of the `read()` system call must have been written for this quasi-invariant. As long as a file is open as read-only, the end of the file will be fixed. This means that the inode will not have to be accessed to check the value of the end of file every time a read is made; that value can be written into a memory location (cached). This

specialization could save disk access time since there is only limited cache space for inodes and an access to the inode may require a disk access.

### *b. Replugging at Run-time*

Guards track the status of quasi-invariants. Based on changes of quasi-invariants, the guards trigger the replugging of specialized system calls (Figure 8). The replugging algorithm changes the function pointer to point to the specialized function that is outside of the kernel.

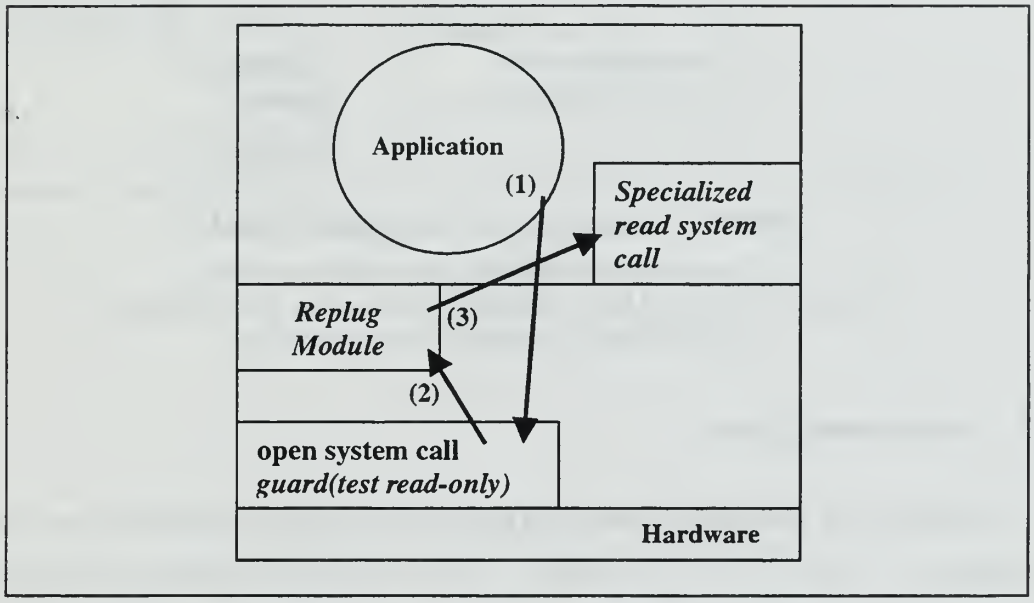


**Figure 8: Replugging at Run-time**

(1) Application opens a file as read-only, (2) guard validates read-only quasi-invariant and triggers replugging, (3) function pointer now points to the specialized read.

### *c. System Call at Run-time*

When an application makes a system call, the operating system evaluates the system call, and uses the modified system call pointer to direct control to the specialized module (Figure 9). The specialized module conducts its work, and returns to the operating system.

The application in our example makes a call to `read()` from the read-only file. The replug module directs this call to the specialized `read()`. The `read()` completes and returns to the operating system.

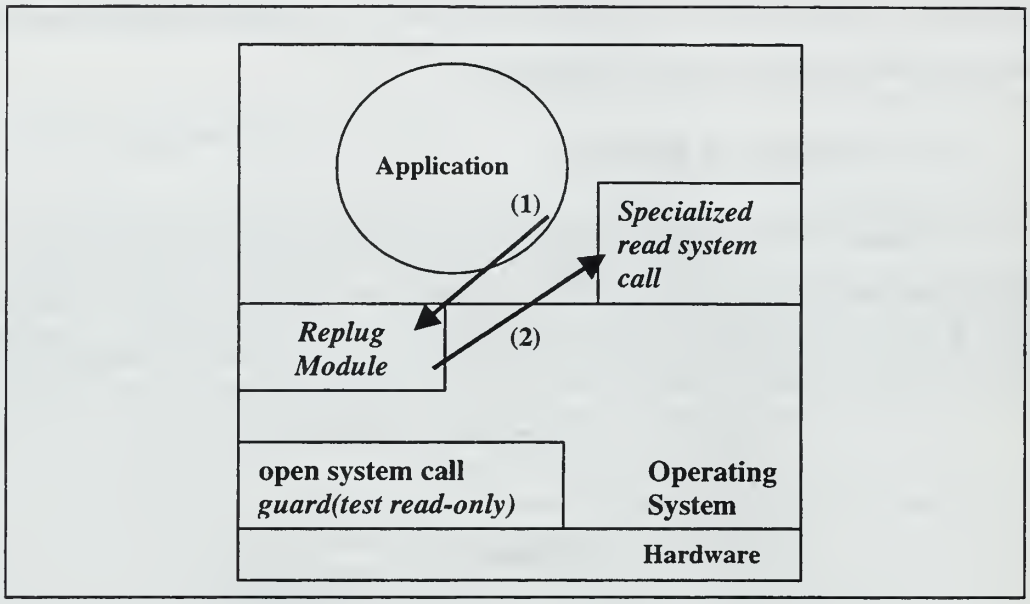


**Figure 9: System Call to Replugged Module**

(1) Application makes a system call to read from the file,
(2) `read()` is directed to the specialized read that does not need to access inode information to check for the end of the file.

### 3. Performance Gains

In testing a fully specialized version of the HP-UX read, the Synthetix team experienced speedup between 1.13 and 3.61 [PUCA96B]. The most significant speedup was experienced in short, sequential reads. Short, sequential reads are one of the most common file accesses in non-database applications, further supporting the use of specialization in this case.

### 4. Observations

Some observations regarding Synthetix are:

- The Synthetix team designed Synthetix for use in customizing an operating system, but the team notes that its techniques may be applied to other programs as well. In fact, we use the idea of specialization to minimize the overhead of our client library.

- Specialized modules reside outside the kernel. It is not clear how such calls are validated for correctness.

## F. SUMMARY

In this chapter we introduced several existing Resource Management Systems and compared them to MSHN. Additionally, we reviewed the techniques developed by the Synthetix team with the goal of applying some of these techniques to reduce the client library's overhead. The next chapter will describe MSHN's client library and its use of system call wrappers to measure resource usage and resource availability.

# IV. WRAPPING SYSTEM CALLS

As an application runs, it uses resources. Examples of these resources include disk space, the CPU and the network. In order to access shared resources in a multi-tasking environment, applications must go through the operating system. When the application needs the resource, it issues a request to the operating system. These requests are to well-defined interface points provided by the operating system, called system calls. Often, the application is linked with a user-level library that it calls. The user-level library, in turn, invokes the system call. For example, in Unix, the user-level `read()` library call invokes `syscall(sys_read, ...)`. [SILB98]

In this chapter we describe our technique for monitoring the resource usage of a user's application. It is based upon intercepting system calls. Intercepting a system call from an application provides a means to gather information on resource usage beyond that provided by the operating system. For example, if we intercept the `read()` library call, we can track the number of bytes read from disk. The technique that we will present in this chapter requires no modification to either the operating system or the user's application, and provides information not otherwise available to the user. We will refer to this technique of intercepting system calls as "wrapping."[4] This chapter will detail the technique for wrapping system calls.

Additionally, we may want to catch the startup and termination of an application. An example of why we might want to do this is to monitor the total run-time of an application. This chapter will present a technique that uses a modified C Run-Time object file and wrapped system calls to catch an application's startup and termination.

## A. WRAPPING SYSTEM CALLS

Operating systems export system call interfaces that applications use to request a resource. The operating system will execute in privileged mode to attempt to grant the requested resource, then return control to the user's code which will execute in user (non-privileged) mode. The Unix operating system calls are often accessed through C language run-time library functions. Each Unix system call has a function in the C library that corresponds to the actual

[4] As stated in Chapter III, the source code of the Condor System is our primary reference for this method [COND96] [LIVN95] [LITZ92] [LITZ97].

system call; we will differentiate this function from the actual system call by calling it the **system call function**. The system call function invokes the appropriate system call using the correct technique (for instance, parameters to the system call may need to be saved in specific registers before the system call is made). Every Unix application program, regardless of the language it is written in, is linked to the C library; the C library effectively becomes part of the application. To application programmers, system calls always look like function calls.

We observe that the code that is executed in privileged mode is not changed unless the operating system is modified. However, it is possible to modify the code that is executed in user mode by intercepting the call to the C library. We will exploit this observation in order to wrap system calls. A basic knowledge of the compilation process is integral to understanding the technique used to intercept system calls; we present a review of this process in the following paragraph.

Quite often, we think of compiling as the process of inputting a file or several files containing source code and outputting executable machine code. However, in most cases, compiling is only the first phase that translates source code to machine code. The second phase, linking, produces the executable file. In the compilation phase, the source code is translated to machine language. As it is translated, all identifiers (e.g., variables, function names) are entered into a symbol table. If a function is defined in a file, it is marked as defined in that file's symbol table, otherwise it is marked as undefined. In the linking phase, the linker tries to find definitions for all undefined symbols. It looks first in object files and libraries specified by the application writer, and then it checks the standard libraries (e.g., the C library, libc.a). The link process is successful when all unresolved symbols are resolved. Once a symbol is defined[5], it may not be redefined. So, if a symbol is defined in a file specified by the application writer, then this original definition would be used in the executable, even if the symbol were also defined in a standard library that is linked after the specified files. This last observation will be important in explaining how we wrap system call functions. [STEV92]

We now present a high level example of the process of creating an executable from source code (Figure 10). The source file shown, main.cc, calls the `read()` system call function. In the compile phase, the compiler creates object code and lists all variables and function names

[5] This implies symbols with global or file scope. A complete discussion of scoping of variables and the effect on the symbol table is beyond the scope of this thesis, however [FISC88] covers this in some depth.

in the symbol table. Since `read()` is not defined within main.cc, it is listed as an undefined symbol. In the link phase, all unresolved symbols must be resolved. No libraries or object files are specified for linking with main.o, so the definition of the symbol `read()` in the standard C library is used to resolve the undefined symbol.

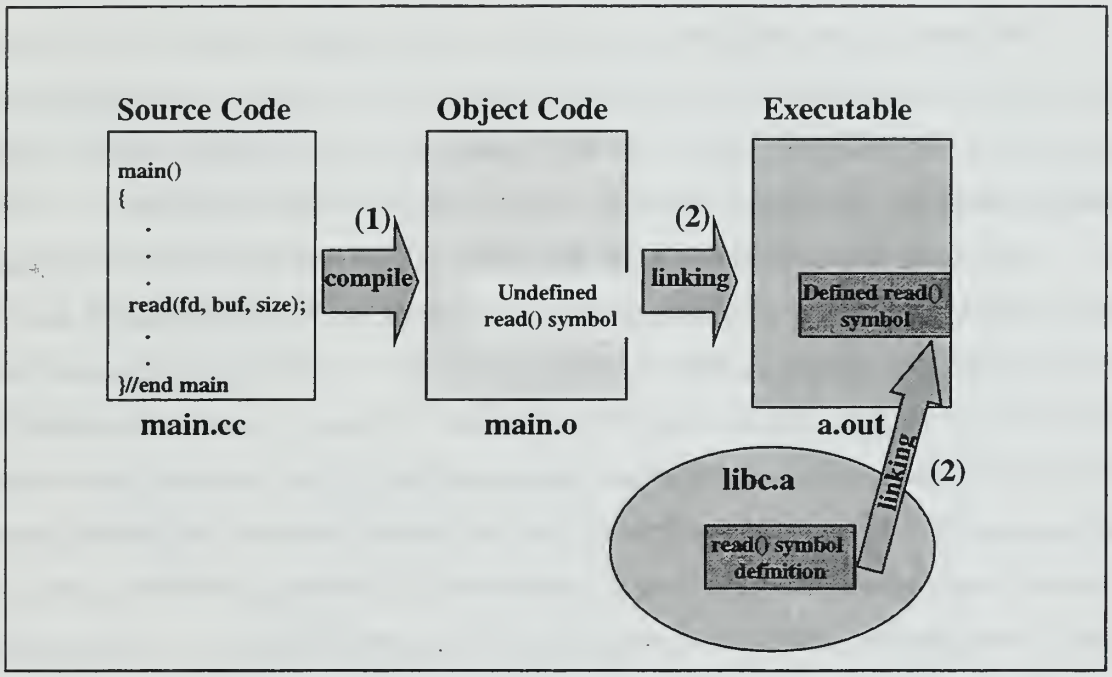


**Figure 10: Creating Executable from Source Code**

(1) Compiling creates object code with unresolved symbols.
(2) Linking resolves the `read()` symbol by linking with the definition in the standard C library.

The steps that we will follow to intercept, or wrap, a system call from an application are:

1. Identify system calls to be wrapped.
2. Modify the C library's names of the system call functions that are being wrapped.
3. Write a wrapper function for each identified system call function. The wrapper will invoke the original system call function by using its new name.
4. Link the wrapper function with the modified C library into a composite library.
5. Link the application with the composite library.

The following sections describe these steps in detail. In order to clarify the steps, we continue the example presented above by writing a wrapper for the `read()` system call function.

### 1. Identify the System Calls to be Wrapped

The first step is to decide which system calls we need to wrap. In Chapter V, we enumerate both operations that need to be intercepted as well as the system calls that need to be wrapped in order to intercept them. In this section, we simply discuss some of the rationale needed to make the list of system calls after the operations have been identified.

Depending upon what information we want to collect, we may need to wrap a number of system calls. For instance, if we want to monitor the number of bytes read across the network, it may first appear that we only have to wrap the `read()` system call. However, this would assume that every `read()` is from the network. Obviously, this assumption is invalid. Therefore, we need a way to differentiate between different types of reads (e.g., network, local disk, remote disk, and standard input). One way to differentiate between different types of reads is to wrap every system call that returns (creates) a file descriptor (fd) that can be read by the `read()` system call, and record the resulting fd number and type in a data structure. The system calls that must be intercepted include `socket()`, `accept()`, `connect()`, `open()`, and `pipe()`. The wrapper for each of these calls will enter the fd number and type into the data structure. On calls to `read()`, we first access the fd data structure and check the type of the fd to be read. Knowing the type, we can discern network reads from other reads, allowing us to track the total amount of traffic sent across the network. This short discussion illustrates the importance of considering dependencies when identifying the system calls to wrap. For simplicity, as we continue our example we will simply describe how to wrap the `read()` system call, irrespective of the type of the file descriptor.

### 2. Modifying Names in the Standard C Library

In addition to accomplishing whatever additional functionality we want in the wrapper, the wrapper must provide the same functionality as the C library's system call function. In order to intercept the call, we create a function with the same signature and symbol name as the system call function. Therefore, we cannot call the C library's system call function to get service from

the operating system. We use an example to clarify this point. Within the redefined function, `read()`, we cannot call the system call function `read()` to get service from the operating system; to do so would be recursive. How, then, may we get the original functionality of the system call function? A Condor paper [LIVN95] suggests one method which is to use the `syscall()` function to pass control directly to the Unix operating system. Invoking `syscall()` with the appropriate system call number transfers control immediately to the kernel. At this point, it is important to note the difference between a system call and the system call function in the C library. Using `syscall()` invokes the system call directly, bypassing all functionality of the system call's C library function. Some of the C library system call functions simply invoke their respective system calls. In these cases, using `syscall()` is fine. However, as the Unix man page for `syscall()` notes, this is not recommended because many of these functions contain important administrative functionality in addition to making a system call [BERK91]. Bypassing this functionality may cause unanticipated behavior or, at the very least, may change the performance that the application expects. For example, when the `exit()` system call is invoked by an application (as a C library function), it calls all registered exit handlers, and then cleans up standard input/output (I/O) by closing all open streams. Once this is complete, it executes the actual system call by trapping to the operating system. If we wrap the `exit()` system call function, and simply call `syscall()` in the wrapper, our wrapped `exit()` would neither call the exit handlers nor perform I/O cleanup. In this case, the application that is wrapped with our library would not get the correct functionality of the `exit()` system call. So, the challenge is to provide an alternate way to perform the system call while not changing the expected functionality of the corresponding system call function from the C library and without completely duplicating the C library functionality.

A solution to this problem, used by the Condor system, is to create a modified version of the C library. We will modify the C library to change the symbol name of the system call function to be wrapped. Thus, the original system call function may be called using a different name. In our example, we will make a copy of the C library, and change the defined symbol `read()` to `READ()`. We will then link this modified library with our wrapper. This provides a way to get the functionality of the system call function, while calling it another name and avoiding recursion.

The steps to create this modified C library, using our example of the `read()` system call function, are (Figure 11):

- Copy the C library, libc.a, into a temporary directory, and unarchive it.
- Modify the symbol names of the system calls that are to be wrapped, e.g., change the `read()` symbol to `READ()`.
- Rearchive this modified C library, e.g., to libModc.a.

After completing these steps, we have a library that contains the systems' original `read()` system call function, with the `read()` symbol name changed to `READ()`.

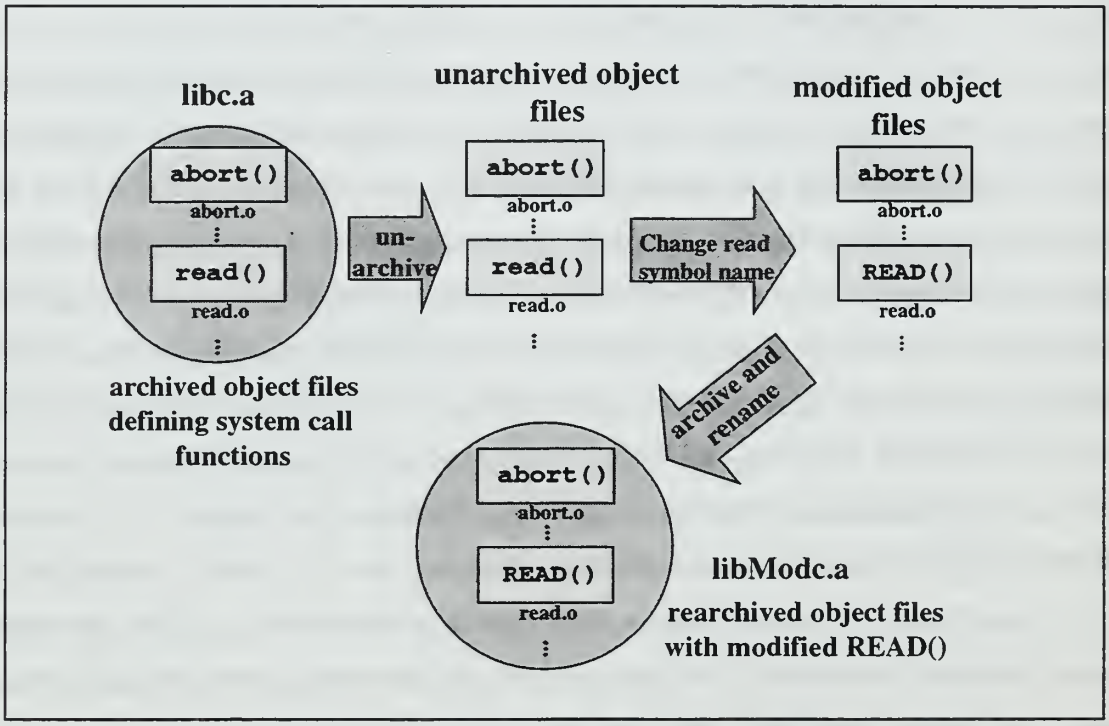


**Figure 11: Creating a Modified C Library**

### 3. Write a Wrapper Function

In order to wrap the system call correctly, we must meet two requirements. First, we must replace the C library's system call function so that, when an application makes a system call, our wrapper is invoked rather than the C library function. To do this, we provide a function that has the same name, return value type, and number and type of input parameters as the original system call function. The objective is to create a function that will have the same signature as the compiled system call function in the C library. This new function will contain

the additional functionality we desire. Referring to our example, Figure 12 shows a wrapper for the `read()` system call function that counts the total number of bytes read, and stores this value in a variable called `sizeOfReads`.

Second, the wrapper must provide a way to call the original system call function. The wrapper must provide the exact functionality that the application would expect from the original system call function. In our example (Figure 12), our wrapper calls a function named `READ()`, which is defined externally, to invoke the original system call function. The derivation of `READ()` was explained above. The keyword `extern` must be used twice in the `READ()` declaration. In the first line, it is used to instruct the C++ compiler to compile the function as a C function, producing a C symbol name. In the second line, the use of `extern` tells the compiler that the definition of the `READ()` symbol will be provided at link time; the compiler lists `READ()` in the symbol table as undefined and does not raise any errors or warnings.

```
//FILE: wrapper.cc

extern "C" {
   extern int READ(int fd, char* buf, int len);
}//end extern "C"

static unsigned sizeOfReads = 0;

int read(int fd, char* buf, int len)
{
   int numBytesRead = READ(fd, buf, len);

   if (numBytesRead > 0){
      sizeOfReads += numBytesRead;
   }//end if

   return (numBytesRead);

}//end read()
```

**Figure 12: Example Redefined `read()` System Call Function**

The `read()` wrapper function is then compiled to create an object file (Figure 13). Recall that were we to look at the symbol table of this object file, we would see the `read()` symbol defined, and the `READ()` symbol undefined; `READ()` is also resolved during linking as described in the previous section.

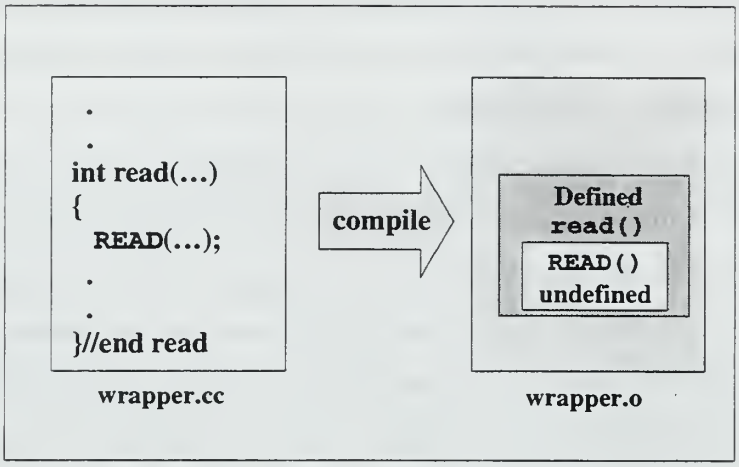


**Figure 13: Compiling the Wrapper**

### 4. Linking the Wrapper Function with the Modified C Library

The next step is to perform an incremental link of the modified C library and the wrapper program (Figure 14).

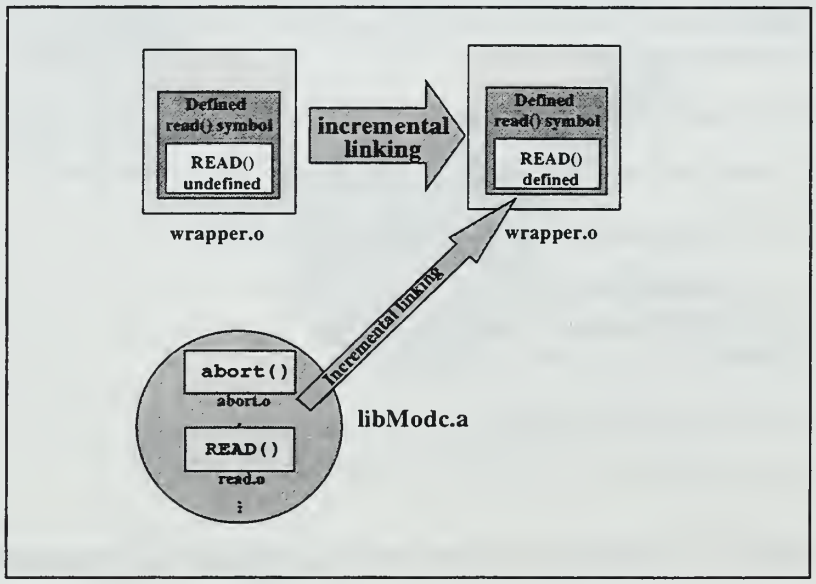


**Figure 14: Linking `read()` Wrapper with Modified C Library**

Recall, from our example, that the `READ()` symbol is still undefined in wrapper.o, and that we have simply renamed the symbol corresponding to the `read()` system call function as `READ()` in libModc.a. The incremental link will use the defined `READ()` from the modified C library to resolve the undefined symbol in wrapper.o, resulting in a more complete object file.

At this point, we have a wrapper object file that can be linked with any application. As long as the required functionality does not change, the steps 1-4 need not be repeated; we only need to create this wrapper once. This simplifies the steps that an application writer must take to wrap his application; he only needs to link with our wrapper object file, and does not have to be concerned with linking with our redefined C library. In the next section we will discuss linking the wrapper with an application.

### 5. Linking the Wrapped System Call with the Application

The last step is to link our wrapper with the application to be wrapped. We return to the example that began this section: taking a simple application that makes a `read()` system call function from source code to executable (Figure 15).

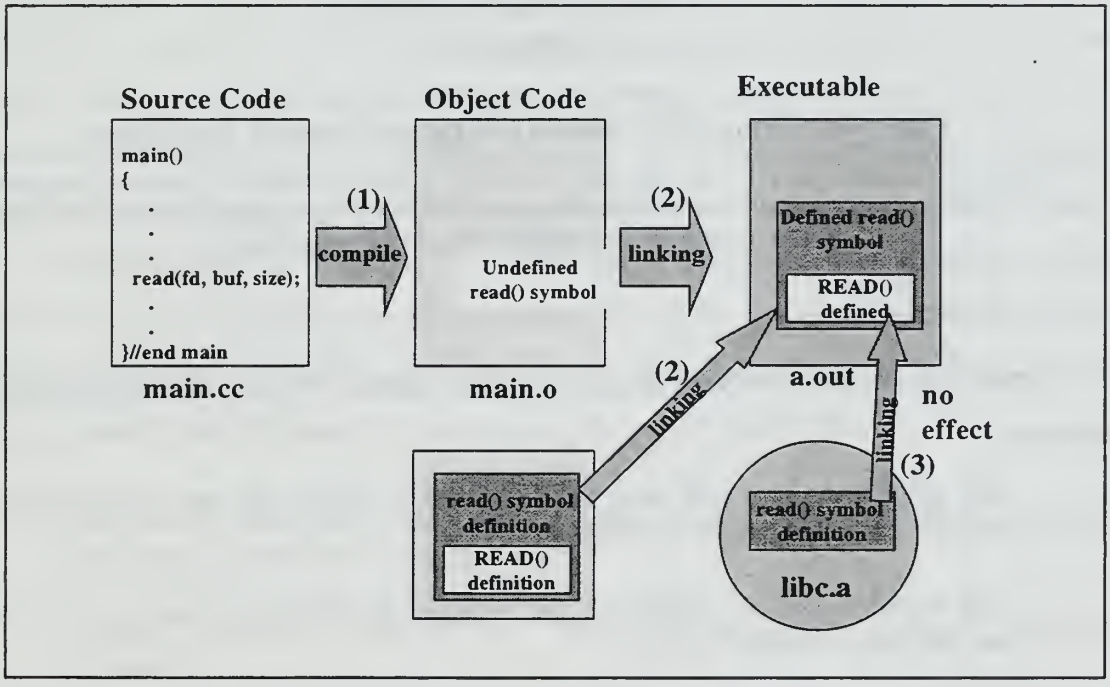


**Figure 15: Compiling a Wrapped Application**

This application, main.cc, is compiled into object code, main.o. In the object code file, the `read()` symbol is undefined. In the next step, the application is linked (explicitly) with the wrapper. This causes the `read()`, called from main, to be defined as the wrapped `read()`. Finally, subsequent linking with the C library, libc.a, has no effect on `read()` because the

`read()` symbol has already been defined. Figure 16 shows, conceptually, how this wrapped `read()` will execute during run-time.

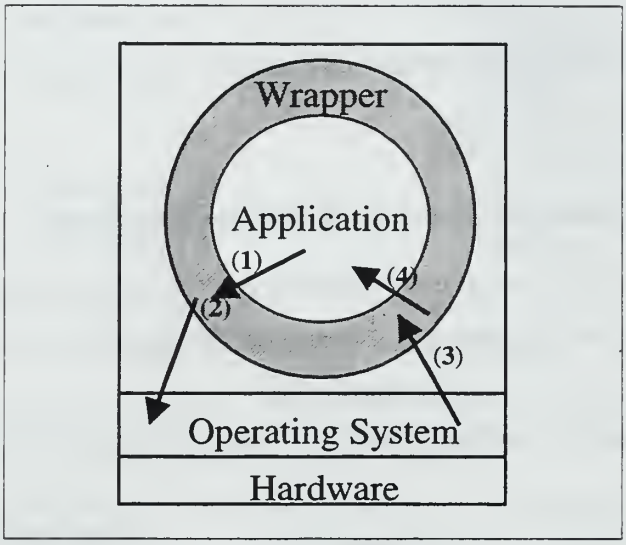


**Figure 16: Conceptual View of a Wrapped `read()` Execution**

(1) Application makes a `read()` system call, (2) `read()` wrapper catches the system call, increments numberOfReads, and passes call on to the operating system, (3) operating system returns number of bytes read, (4) wrapper returns number of bytes read to application.

### 6. Summary

Listed below are some observations about this method of using wrapped system calls in this manner:

- Any application that both calls `read()` and is linked with our wrapped system call will use our redefinition of `read()`.
- Any library function invoked by the wrapped program that uses `read()` will also use our newly defined `read()`.
- No modification to application source code or to the operating system is required.
- `READ()` has the full functionality of the original C library `read()` because it is the original C library `read()` – it has simply been renamed.
- No change has been made to the system's original C library. We simply modified a copy. So, a developer who does not want to have his application linked with the wrapper would simply compile and link as he would normally.

- Finally, wrapping system calls in this manner gives access to information not otherwise available or easily accessible. We will discuss this further in the next chapter.

## B. CATCHING STARTUP AND TERMINATION OF AN APPLICATION

In the previous section, we discussed the process of wrapping system calls that are invoked by a running application. However, as motivated above, we also want to perform particular actions upon the startup and termination of an application. In this section we will discuss how to use a modified C Run-Time object file, crt0.o or crt1.o (C Run-Time 0 or 1), to catch application startup. We will look at alternative methods for catching application termination.

By default, every application is linked with several files and libraries; one of these files is the C Run-Time object file. The C Run-Time file is the start up routine that is called to start an application. Based upon the version of operating system, program language and compiler, the exact functionality of the C Run-Time object file can vary. However, generally it performs the following tasks at a minimum: initializes the stack, initializes the heap, calls constructors, calls `main()`, and calls `exit()` [STEP92]. In a manner similar to how we wrapped system call functions in the previous section, we can wrap the C Run-Time call to `main()`. This allows us to catch the startup and termination of an application. The Condor System used this method in restarting a checkpointed program; we list the steps they used [COND96].

1. Copy the C Run-Time object file (in most systems crt0.o or crt1.o) to a working directory and rename it, e.g., mod_crt0.o.

2. Change the symbol name in the C Run-Time object file, `main()`, to `MAIN()` (Figure 17).

3. Write a function, `MAIN()`, that performs whatever additional tasks you want to do at startup. It must also call `main()`.

4. When linking an application to create an executable, explicitly replace the command to link with the system's C Run-Time object file with one that causes a link to the modified C Run-Time object file. Additionally, link the application with the file that contains the `MAIN()` function.

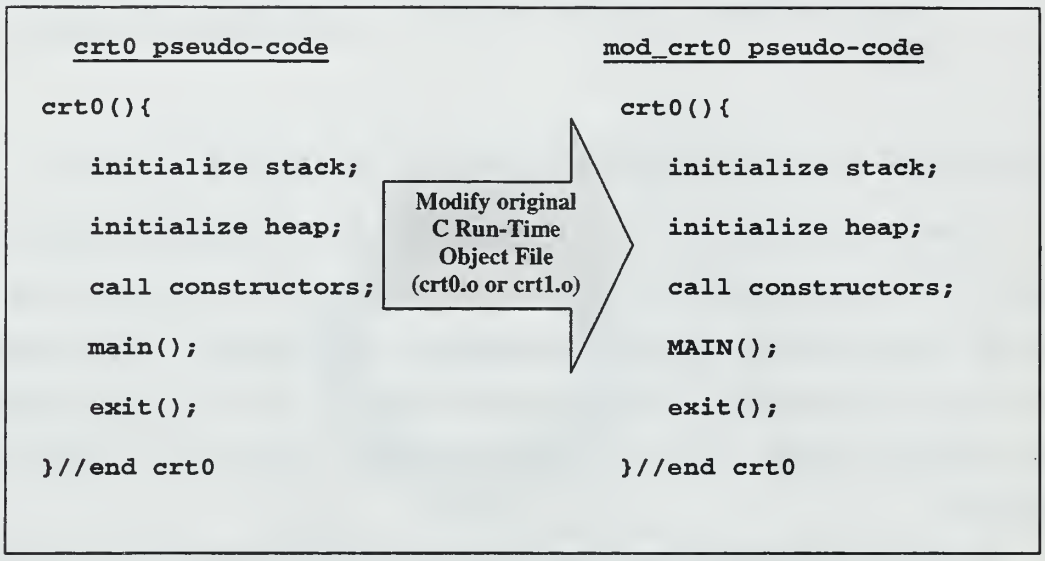


**Figure 17: Pseudo-code for C Run-Time File**

When the application is started, the modified C Run-Time object file invokes `MAIN()` rather than `main()` (Figure 18).

```
//FILE: MAIN.cc

extern "C" {
   extern int main(int argc, char* argv[]);
   int MAIN(int argc, char* argv[]);
}//end extern "C"

int MAIN(int argc, char *argv[]){

   setStartTime();

   int returnVal = main(argc, argv);

   calcRunTime();

   return returnVal;

}//end MAIN()
```

**Figure 18: Example Code for `MAIN()`**

`MAIN()` will execute the additional code that we wrote, then call `main()`. Upon completion of `main()`, control will return to `MAIN()` and the application will exit. If we insert

code before `main()` is invoked by `MAIN()`, we in fact "catch" the start of the application. Likewise, if we insert code after `main()` returns, we effectively catch the termination of the application, if the application terminates normally. For example, if we want to determine the amount of wall clock time that it takes an application to run, we can insert code to get a clock reading before `main()` is called and after it returns, and then compute the difference to find the total time. However, in the case of an abnormal exit from `main()`, control will not return to `MAIN()`. In our example, if control never returns to `MAIN()`, we would never have caught the termination of `main()` and the total time would not be calculated. For this reason, we will use this technique only to catch the startup of an application.

Another option for catching the end of a process is to register a function to be called on `exit()` using the `atexit()` function. In most implementations, `atexit()` can register up to a fixed number of functions to be performed upon application termination (32 in ANSI C) [STEV92]. However, using this approach takes away one of the `atexit()` functions available to the application writer. Additionally, though not advised, an application writer could directly call the `_exit()` system call function which directly invokes the system call without calling the `atexit()` functions. In this case, whatever functionality we wanted upon application termination would be bypassed. Finally, not all implementations of Unix implement the `atexit()` function. For these reasons, we will not use this technique to catch the termination of an application.

The last option that we address is to wrap both the `exit()` and `_exit()` system call functions using the method described above in Section A. It is important to note that when a process terminates, unless it specifically calls `_exit()`, it will call `exit()`. Therefore, by wrapping `exit()` and `_exit()` we catch the termination of the process, even if the application exits abnormally. We use this method in MSHN for catching the termination of an application.

## C. ALTERNATIVE TO WRAPPING SYSTEM CALLS

We have reviewed one technique for wrapping system calls that requires object code. Another team at the University of Wisconsin-Madison has developed tools that could lead to an additional approach that simply requires executable code. The Paradyn project and its predecessor, the Executable Editing Library (EEL), provide tools to modify the compiled and

linked executable code of an application. The EEL toolkit enables analysis and modification of an executable statically (before run-time), while Paradyn allows the analysis and modification of executing code. Using these tools, we could modify the C library functions as they are defined in the executable code, effectively wrapping system call functions. [PARA97] [LARU97]

When using the EEL, we would encapsulate the additional functionality that we want from each system call function wrapper as a "snippet" (or subroutine). The snippet is written in a high-level language such as C++ and then is compiled to assembly language. The EEL's Control-Flow Graph (CFG) tool allows manual editing of a routine within an executable program; in our case, we would use the CFG to include our snippet in the system call function that we want to modify. This technique would allow us to wrap system call functions without having access to source or object code.

## D. SUMMARY

This chapter presents a mechanism for wrapping system calls and a technique for catching the start and termination of an application. The following chapters will discuss how we monitor an application's use of resources, and how we measure the load on resources that can be scheduled by MSHN.

# V. RESOURCE MONITORING BY THE CLIENT LIBRARY

In the previous chapter, we described mechanisms for intercepting calls made by applications to the operating system. Intercepting and inspecting parameters of these system calls provides one way to monitor the resource usage of an application. The MSHN scheduling advisor requires an estimate of resources to be used by an application in order to wisely schedule the application. This chapter enumerates the information that may be needed by the MSHN scheduling advisor, determines what information is already reliably available via queries to the Unix operating system, and enumerates the system calls that must be wrapped to obtain the remainder of the information.

## A. RESOURCE USAGE DATA REQUIRED BY THE MSHN SCHEDULER

MSHN requires accurate data on resource usage information for applications that run within the MSHN system. The MSHN scheduler uses this information to make scheduling decisions. In our system, the client library gathers this data as an application runs and, upon termination, sends this information along with the machine identification and application name to the Resource Requirements Database (RRD). The RRD filters this data and stores it for later use by the scheduling advisor in estimating requirements for future runs of the same application.

Our first requirement is to determine what resources we must monitor. Table 1 lists all of the resources that we have identified, and metrics associated with each of them. We list metrics because many ways may exist to measure the use of any particular resource. Disk usage provides a good example of this. Possible ways to measure disk usage include total time waiting on input or output, total bytes read or written, total number of read or write calls, and time between requests. For each of the resources identified in Table 1, we have selected the metrics that we believe will best aid the scheduling advisor in scheduling an application the next time that it executes. We expect that research now in progress [CARF99] on the granularity of information needed by the Scheduling Advisor (SA) may limit or slightly modify this list.

With the increase in mobile computing, the power that an application requires to run may be critical in deciding whether to schedule it locally or remotely. An application that may be able to run most optimally locally, considering everything except battery power, may still be most wisely scheduled to run remotely if it would consume excessive power if executed locally.

| Resource | Metric |
|---|---|
| Total Run-time | Time |
| CPU | Time |
| Memory | Maximum memory used |
| Cache Memory | Maximum cache used |
| Local disk | Bytes read |
| | Number of reads |
| | Bytes written |
| | Number of writes |
| Network disk | Bytes read |
| | Number of reads |
| | Bytes written |
| | Number of writes |
| Network | Bytes read |
| | Number of reads |
| | Bytes written |
| | Number of writes |
| Local inter-process communication | Bytes read |
| | Number of reads |
| | Bytes written |
| | Number of writes |
| Keyboard input | Number of bytes |
| | Time blocked waiting for user input |
| Power Consumption | Watts |

**Table 1: Resources to Monitor**

Total run-time (or wall clock time) is perhaps the most obvious measure of resource usage; it answers the most basic of questions, "How long does it take for this job to run?" SmartNet used this measure to update each application's ETC entry in the SmartNet Database. However, we learned in our experience with SmartNet that this metric alone is not sufficient. Using only run-time requires an application to have executed in every conceivable configuration in order for the scheduling advisor to consider mapping it to all available machines. Therefore, the MSHN scheduling advisor will consider a more comprehensive list of resources required and available when making a schedule.

We sub-categorize the non-power resources as non-I/O and I/O resources. In the non-I/O category are CPU, memory and cache memory utilization. For CPU utilization, we will use total

CPU time as the metric that describes how extensively the CPU is used by an application. For both memory and cache memory, we will use maximum memory size used by the application to run.

The I/O resource category includes local disks, network disks (e.g., network file system (NFS)), terminal I/O, interprocess communication, and other network traffic. Because I/O can be so time consuming, it is important to measure each processes' use of these resources. The metrics that we will use in each case are the total size of the reads and writes to each resource, and the total number of accesses to each resource. Additionally, for terminal I/O we monitor the total time a process spends blocked waiting on user input.

In the following paragraphs, we discuss the methods that we used to gather the resource information described above.

## B. INFORMATION AVAILABLE FROM USER-LEVEL OPERATING SYSTEM QUERIES

Some operating systems provide system calls that can be used to query the type and amount of resources used by a process. In this section we discuss information that can be reliably determined via user-level queries in various versions of Unix. One call that is available in most versions of the Unix system is `getrusage()`. Another widely available method for determining resource usage on a Unix system is by examining the contents of the `/proc` directory.

One challenge that we faced in using native Unix methods is the variation in their functionality. This variation is due to a number of factors, including system configuration and differences in implementation from one operating system release to the next. Using `getrusage()` as an example (Table 2), we see that the implementation of this C library function varies widely with the version of the operating system. This table shows us that the implementation is not consistent across versions of the operating system. Further, although one would expect that moving to a more recent version of an operating system would at least maintain existing functionality, this is clearly not the case as illustrated by the loss of substantial functionality in the move from Sun 4.1.4 to Sun 5.5. Making this situation more difficult is the fact that differences in functionality are not well-documented in all cases. The Sun 4.1.3 and

4.1.4 man pages imply that their `getrusage()` implementation does account for block input and output operations and context switching.

| Selected getrusage() fields | Operating System And Version | | | | |
|---|---|---|---|---|---|
| | Linux 2.0.29 | Sun 4.1.3 | Sun 4.1.4 | Sun 5.5 | Sun 5.6 |
| | implemented? | | | | |
| User CPU time | yes | yes | yes | yes | yes |
| System CPU time | yes | yes | yes | yes | yes |
| Max resident set size | yes | yes | yes | no | no |
| Unshared data size | no | yes | yes | no | no |
| Shared memory size | no | no | no | no | no |
| Minor page faults | yes | yes | yes | no | no |
| Major page faults | yes | yes | yes | yes | yes |
| Block input operations | no | yes | yes | no | no |
| Block output operations | no | yes | yes | no | no |
| Voluntary context switches | no | yes | yes | yes | yes |
| Involuntary context switches | no | yes | yes | yes | yes |

**Table 2: Comparison of `getrusage()` Implementations**

Because `getrusage()` could not be reliably used to obtain other information, we used it only to measure CPU time. Unfortunately we cannot use it to obtain the maximum resident set size (maximum size of memory used) of a process because, as Table 2 shows, the `getrusage()` supplied with Sun operating system versions 5.5 and 5.6 does not have the maximum resident set size implemented. Therefore, for this information, we would need to directly read from the `/proc` file system or use the output of the `ps` utility (an active monitoring technique, see Chapter VI) to obtain this memory utilization information.

Another method for obtaining information about resource usage from a Unix system is to examine the contents of files in the `/proc` directory. The following information may be available from those files: CPU load, memory availability and utilization, network packet information, and page fault information. Unfortunately, whether or not this information is actually maintained by the system appears to be a factor of system administration. We have a Linux PC in our lab that was maintained by the Linux special interest group (SIG) experts and which appears to function reliably to all students and faculty who used it. However, when we examined the contents of the `/proc` files after executing some programs which should have caused them to be updated, we found that they were not. We encountered an additional problem when trying to use `/proc` to examine system-level information as an alternative to monitoring

each and every application to assess resource usage. The problem involved gaining access to the `/proc` directory. We found that while we could access process-level information on all systems, and system-level information on our Linux platform, we could not access system-level information on the Sun platforms; access was restricted to the privileged user (kernel). Because of these problems, we therefore must conclude that the MSHN RRD and Resource Status Server (RSS) cannot depend upon obtaining information from this source.

## C. WRAPPING SYSTEM CALLS TO MONITOR RESOURCE USAGE

We use wrapped system calls in monitoring resource usage in four ways: (1) to directly monitor resource usage; (2) to trigger the use of specific resource monitoring utilities; (3) to calculate total run-time; and (4) to calculate resource availability and update the RSS. We will discuss the first three of these purposes in the following paragraphs; the fourth will be discussed in Chapter VI.

### 1. Using Wrapped System Calls to Directly Monitor Resource Usage

We use our wrapping technique primarily to measure resource usage related to input and output (I/O). There are several reasons for this. First, input and output operations are costly in terms of time and cannot be overlooked. Second, the resource usage information that we need is not available at the granularity that we need through existing utilities. Finally, as addressed above, implementation is not consistent across platforms. In this section, we will address measuring I/O resource usage which is the most complex resource usage to measure. Code for other system call wrappers is available at the MSHN web site at http://www.mshn.org.

Our first challenge is to differentiate between the various types of input and output. Unix generalizes I/O by making all output appear as though it is to a file. For example, the system call, `open()`, that prepares a file for I/O operations returns a file descriptor. The file descriptor is an integer value that uniquely identifies an I/O device to the process. Other system calls that prepare a device or connection for I/O, for instance `socket()`, also return file descriptors. This simplifies the programmer's view of input and output; generally, all file descriptors can be treated the same. However, while this simplifies the programmer's job, it makes our task more challenging because, as discussed in the previous chapter, when a `read()` is intercepted it is not obvious how to determine the type of the file descriptor parameter. We therefore implement a

technique using system call wrappers and a data structure to allow us to differentiate between file descriptors.

Table 3 lists the system call function wrappers that we use to catch the initialization of all file descriptors. Each of these wrappers enters the resulting file descriptor into a hash table data structure, `fdTable`. Also, some of these wrappers need to do additional checking to differentiate between file descriptor types. For example, the `open()` system call can open files that may be local or may be mounted over a network file system. When we access the `fdTable`, we need to know whether a file descriptor is for a local or remote file.

| System Call Function | Functionality of Wrapper |
|---|---|
| `open(), __open()` | Determines whether opened file descriptor is local or remote (from a remote file server) and enters file descriptor and appropriate type, LOCAL_FILE or REMOTE_FILE in fdTable. |
| `socket()` | Determines whether socket is Unix or Internet. Enters file descriptor and corresponding type, LOCAL_IPC or NETWORK_IPC, into the fdTable. Also enters this machine's IP address into fdTable. |
| `accept()` | After accepting a connection, evaluates the address of connected host to determine whether this process is on the same machine as the connecting process. If so, enters file descriptor and LOCAL_IPC into fdTable. Otherwise, enters file descriptor and NETWORK_IPC into fdTable. |
| `connect()` | After connection accepted, evaluates address of connected host to determine whether this process is on the same machine as the accepting process. If so, updates file descriptor's type entry in fdTable as LOCAL_IPC. Otherwise, updates file descriptor type entry in fdTable as NETWORK_IPC. |
| `close(), __close()` | Removes the file descriptor's entry from fdTable. |
| Other calls that create file descriptors | Enters file descriptor into fdTable with file descriptor type of LOCAL_IPC. |

**Table 3: Wrappers for System Call Functions that Return File Descriptors**

The data structure that we have implemented for `fdTable` is a hash table that uses chaining for collision resolution [CORM97]. Each element of the structure is indexed by hash value that is derived from the file descriptor number. Each element contains data fields for file descriptor, file descriptor type, IP address, as well as fields for other information required to estimate latency and throughput (discussed in Chapter VI). This hash table structure is constructed such that entry, search, and deletion of elements from the structure are made in

constant time (that is, O(1)).[6] Further, the access functions are constructed such that retrieval is the fastest of these three operations. Use of this structure minimizes the overhead associated with keeping this additional information on file descriptors.

Having wrapped the system calls that create file descriptors so that file descriptor types can be retrieved quickly given only the file descriptor, we can now examine I/O system call function wrappers. The first wrapper that we will review is the `read()` system call function wrapper. Each time an application, or a library function linked with the application, calls `read()`, the `read()` function wrapper will be invoked as discussed in Chapter IV. An event flow diagram is provided in Figure 19.

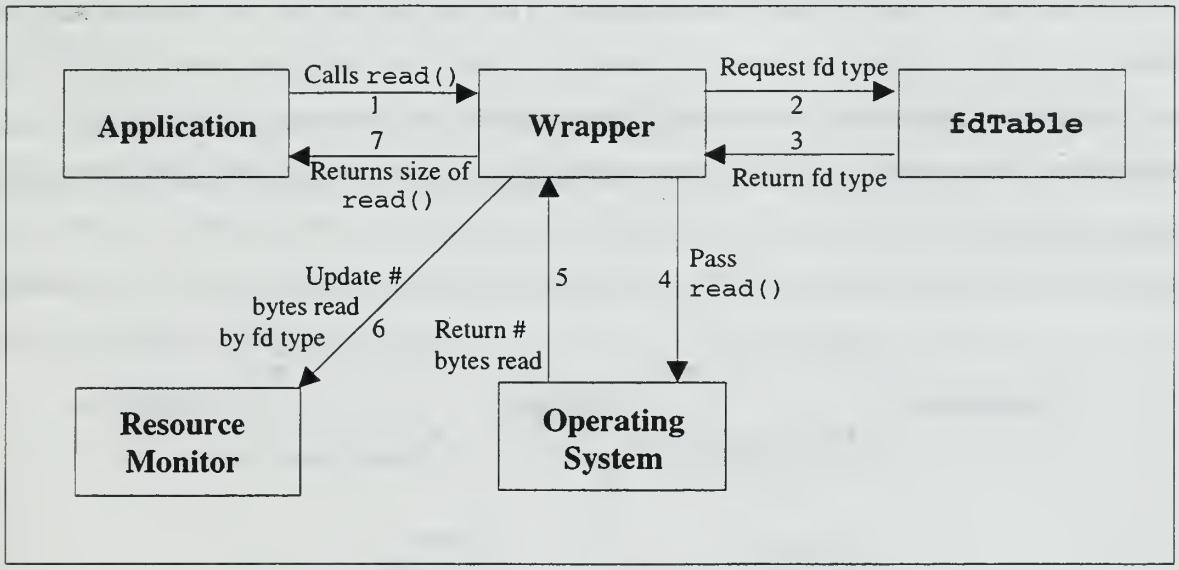


**Figure 19: Event Flow Diagram for `read()` invocation**

The application's call to `read()` is intercepted by the wrapper. Using the file descriptor that was passed as a parameter in the application's `read()` call, the wrapper requests the file descriptor type (LOCAL_DISK, REMOTE_DISK, NETWORK_IPC, LOCAL_IPC or TERMINAL). The `fdTable` returns the file descriptor type to the wrapper. The wrapper passes the `read()` call on to the operating system. When the operating system finishes servicing the `read()` call, it returns the total number of bytes read to the wrapper. The wrapper uses this information and the file descriptor type to update the resource monitor. We note that

[6] In order to assure that search time is O(1), the size of the hash table must be at least proportional to the maximum number of file descriptors to be entered into the table [CORM97]; we provide a variable that can be used to accomplish this resizing at compile time.

the resource monitor is not the MSHN RRD. Rather, it is an object local to the process that keeps a running record of all resources used by the application. Additionally, it provides methods for updating the RRD with this information on application termination. Finally, the wrapper returns the size of the `read()` to the application.

The `write()` wrapper is similar to `read()`, as shown in Figure 20. The application's call to `write()` is intercepted by the wrapper. Using the file descriptor that was passed as a parameter in the application's `write()` call, the wrapper requests its type. The `fdTable` returns the file descriptor's type to the wrapper. The wrapper passes the `write()` call on to the operating system. When the operating system completes servicing the `write()` call, it returns the total number of bytes written to the wrapper. The wrapper uses this information and the file descriptor type to update the resource monitor. Finally, the wrapper returns the size of the `write()` to the application. In actuality, there is additional functionality in both our `read()` and `write()` wrappers to measure end-to-end perceived QoS. That functionality is the topic of the next chapter.

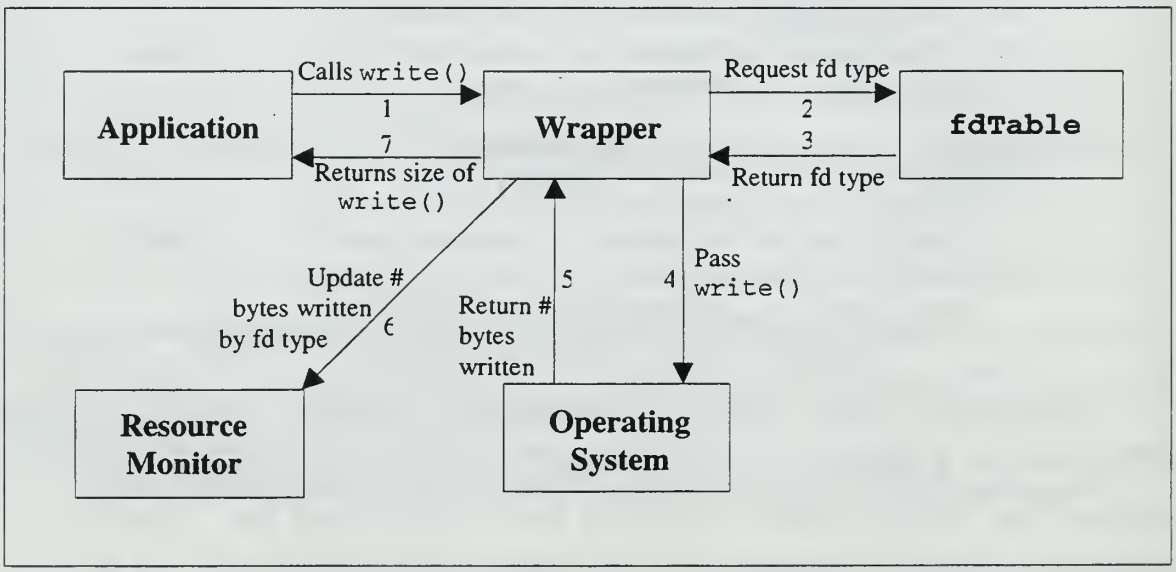


**Figure 20: Event Flow Diagram for `write()` Invocation**

### 2. Triggering Calls to System Utilities

Earlier in this chapter, we indicated that we use `getrusage()` to obtain CPU time. We use our wrappers of the `exit()` and `_exit()` system call functions to catch the termination of

an application. At application termination, our `exit()` and `_exit()` system call function wrappers call `getrusage()` to obtain total CPU time.

### 3. Calculating Total Run-time

We calculate total run-time in the same way as described in our example in Chapter IV. We modify the C Run-Time object file to record application start time. We catch termination of the application by wrapping the `exit()` and `_exit()` system call functions. At termination, we record the application's termination time and take the difference as total application run-time (wall-clock time).

## D. SUMMARY

This chapter enumerates the resource usage information that may be needed by the MSHN scheduling advisor, determines what information is already reliably available via queries to the Unix operating system, and enumerates the system calls that must be wrapped to obtain the remainder of the information. In the next chapter, we will discuss an approach for passively monitoring network QoS.

# VI. MEASURING PERCEIVED END-TO-END QUALITY OF SERVICE

To support adaptive applications, the MSHN Scheduling Advisor requires end-to-end status information concerning the resources at its disposal. One critical resource is the network; this chapter will address monitoring the current network availability. We first review several tools, protocols and application components that estimate network availability and then summarize the desirable characteristics of these systems. Finally, we present our approach for estimating network availability.

## A. SYSTEMS THAT ESTIMATE NETWORK AVAILABILITY

To support MSHN's end-to-end network monitoring, we initially considered eight application components, protocols, and tools from both the commercial and research sectors: `ping`, `ftp` (the File Transfer Protocol application), Netscape Communicator, Network Weather Service (NWS), Netperf, BBN's Communications Server (Commserver), Resource Reservation Protocol (RSVP) and Simple Network Management Protocol (SNMP).

We quickly rejected two as inappropriate for MSHN: SNMP and RSVP. We could not directly use the Simple Network Management Protocol (SNMP) in MSHN because it provides link-based information and does not estimate end-to-end throughput and latency between machines on remote, nonadjacent networks [PERK97]. Any tool built on top of SNMP would necessarily have to change when routing algorithms changed. We also considered RSVP, but have rejected it largely due to recent results indicating that the bandwidth it allocates can be substantially different from that requested [LEEC98]. We now discuss each of the six remaining in detail and then compare and contrast advantages and disadvantages of each type.

### 1. Ping

The `ping` program is ubiquitous. Its primary use is for troubleshooting networks. In its default configuration, if a network connection exists between two machines, the execution of `ping` results in a short message making a round trip from the local to the remote host and back. When `ping` completes, it prints the number of bytes sent and the round trip time. Thus, `ping` provides a single packet measurement of network throughput.

The `ping` program packages its local timestamp in an Internet Control Message Protocol (ICMP) packet and sends it to the targeted host. The targeted host receives this datagram, its IP layer recognizes the type (ECHO_REQUEST) and immediately repackages the data contained in the packet, sending it back to the pinger. The pinger receives the reply datagram and subtracts its machine's current time from the timestamp that the pinger placed in the original datagram, thus determining round trip time. [BERK91]

### 2. File Transfer Protocol

The File Transfer Protocol application (ftp) is used to transfer files between machines connected by a network. We studied ftp because it estimates end-to-end (meaning application to application) throughput after completing a file transfer. The ftp application is a client-server application; the ftp server runs as a background process listening on a fixed port for client connection requests. The user starts the ftp client in order to issue requests to the ftp server on the remote machine. Because of the tight coupling between the ftp server and client, it is possible for ftp to estimate the throughput associated with transferring a file. We now summarize the actions that occur in transferring a file, F, from the ftp server on computer A to the ftp client on computer B.

- The `ftp` server on computer A listens on port Y.
- The `ftp` client on computer B connects to the server on computer A at port Y.
- The server on computer A accepts the connection and generates a child process that will handle all future interactions with the `ftp` client via the connection. We will call this connection the **control connection**.
- The `ftp` client on computer B sends a request across the control connection to the `ftp` server on computer A to send file F. Included in the request to A is the port number, X, that the client on computer B will listen to.
- The `ftp` client on B then listens on port X.
- The server sends the size, S, of file F to the `ftp` client over the control connection.
- The client receives S.
- The server connects to the client on computer B at port X.

- The client accepts the connection and records computer B's time as $T_{startRead}$. We will call this connection the **data connection**.

- Once the server sees that the client accepted the data connection, it sends the file, F, across that connection and the client reads until it receives the entire file.

- When the entire file has been received, the client records computer B's time as $T_{endRead}$. The client then uses S and the time elapsed between $T_{startRead}$ and $T_{endRead}$ to estimate throughput.

### 3. Netscape Navigator

Netscape recently made the source code for their web browser freely available [NETS98]. By examining this source code, we learned that they use an approach similar to `ftp`'s to calculate throughput. Netscape Navigator displays a "thermometer" at the bottom of the browser, showing the user the current download speed, the amount of the file already downloaded, and the estimated time to complete the download. When downloading large files, the system call `read()` must typically be invoked more than once by the browser. Following each invocation of `read()`, Netscape updates the thermometer's data fields. The throughput estimate displayed on the thermometer is cumulative in that each calculation is based upon the total number of bytes downloaded, the total size of the file to download, and the total time since the first `read()` was called for the current file.

### 4. Network Weather Service

The Network Weather Service (NWS) is a tool for predicting computer and network performance for use by metacomputing applications [WOLS97] [SPRI97]. NWS makes periodic estimates of availability of resources for which it is responsible. One of the estimates made by NWS is network availability, specifically, the latency and throughput observed between two computers.

In order to measure the latency between two computers, A and B, a NWS process on computer A sends a small message to a corresponding process on computer B. The process on B immediately replies to the process on A, with the process on A recording the round trip time. NWS approximates latency as half of this round trip time.

To estimate throughput, the NWS process on host A sends a large message to the corresponding process on host B, and the process on B sends a small acknowledgement message back to the process on A. The NWS process on computer A estimates the transmission time of the large message as the round trip time, less the latency estimate described above. Throughput is then estimated as the message size divided by estimated transmission time. NWS keeps a record of previous estimates of throughput and latency that it uses to predict future resource availability using statistical modeling techniques.

The developers use a token passing scheme to avoid overloading the network. They note that token passing does not scale well with large distributed systems. Therefore, the accuracy of the throughput and latency estimates degrades as the size of the network increases. Additionally, token passing can introduce security problems [STAL98]. Finally, to capture fluctuations in network QoS, the developers note that administrators must increase both message size and monitoring frequency.

### 5. Netperf

Netperf is a benchmark that can be used to measure different aspects of network availability with a primary focus on actively measuring the throughput of bulk data transfers and the request/response round trip time [HEWL96]. Netperf contains a rich benchmarking suite with many options for simulating specific scenarios (e.g., http protocols). Netperf's bulk data transfer test can be used to estimate the throughput between a local and remote host communicating over a network. It works by sending data for a period of time (the default is 10 seconds), and then measuring the total amount of data sent and received after the time period has elapsed. Netperf's request/response time test is very similar to that used by NWS to estimate latency: a short message is sent from a local to a remote host; the remote host replies immediately; and the local host measures round trip time and approximates latency as one half of this round trip time.

### 6. BBN's Commserver

The Joint Task Force Advanced Technology Demonstration (JTF ATD) strives to predict trends in the advances of future hardware and software. It also provides a reference architecture into which such advances can be easily integrated. At the base of the JTF architecture is the

Commserver whose ultimate purpose is to permit applications to be network aware. The job of the Commserver is to estimate the available bandwidth between JTF users. The Commserver uses that bandwidth, along with the priorities of various users (expressed as currency), and a list of the applications requiring execution to allocate resources. [HAYE94]

Early experimentation with the JTF ATD Commserver revealed several problems. First, the Commserver places a load on the network in order to estimate end-to-end latency and bandwidth available [KRES98]. Second, it uses the throughput and latency estimates directly, without reference to previous measurements, to estimate network load. Due to the rapidly changing nature of network traffic, this technique can return inaccurate or misleading results. Finally, the estimates returned are inaccurate unless the network is sampled frequently which further increases the network load.

### 7. Characteristics of These Systems

We divide the mechanisms described above into two categories: **passive** and **active**. Active mechanisms place additional loads on the resources that they are monitoring; passive ones do not. Applying this definition, we see that `ping`, NWS, Netperf and BBN's Commserver all use active mechanisms, while `ftp` and Netscape use passive mechanisms. Unfortunately, it is when the network is most busy that we need the most accurate estimates. This is when there may be no extra bandwidth available to give to these active mechanisms. Passive mechanisms, on the other hand, do not add to the load carried by the already scarce resource. For this reason, in MSHN we prefer passive monitoring.

Another way of categorizing the previously discussed tools and application components for measuring network performance attributes is to consider how closely tied they are to applications. The programmers of both `ftp` and Netscape used domain-specific knowledge to obtain estimates of network throughput. MSHN prefers that application writers not be required to also worry about measuring resource availability; availability should be measured by the system.

Finally, there are sources of error and limitations associated with the previously described mechanisms. When estimating throughput, all mechanisms start timers with a handshake. The best ones then subtract off some multiple of an estimate of latency, which they assume to be the amount of time required for the handshake, but which may actually be substantially different due

to operating system CPU scheduling policies. Further, none of the passive monitoring techniques estimate latency, only throughput. The MSHN system requires, at a minimum, the knowledge of both of these.

Based on these observations, we sought a passive mechanism that would accurately estimate both bandwidth and latency without requiring the application programmer to implement monitoring code. In the remainder of this chapter we describe such a mechanism.

## B. A PASSIVE APPROACH FOR MONITORING NETWORK PERFORMANCE

Before describing our domain-independent mechanism for passively obtaining accurate network performance estimates, we enumerate some of the challenges we faced in evolving such a mechanism.

### 1. Challenges

In order to avoid modifying either the operating system or the system libraries, we chose to apply a technique developed by Condor [LIVN95]. That is, we chose to implement an additional library that intercepts `read()` and `write()` calls and then link applications' object code with that library.[7] This additional library will then be able to pre-process parameters of `read()` and `write()` and post-process the results. We will define this interception of system calls as **wrapping** system calls.

After identifying the mechanism required to implement "domain-independence" and "passive monitoring" we turned our attention to the problem of accurately estimating bandwidth and latency. In the remainder of this section we refer to a process that issues a `write()` call to write across the network as the "writer" and the process that issues the corresponding `read()` as the "reader." We also assume that the reader and writer are using TCP.

In order to estimate latency, we must know when the writer writes the message, and when the reader's computer receives it. We face several problems in trying to accurately obtain these times. First, the reader does not know when the writer wrote the message to the network. Second, if we modify `write()` so that the writer appends its local time to the beginning of the message, we still must compensate for the clock offset between the reader and writer computers.

---

[7] If object code is not available, techniques developed in Paradyn [PARA97] [LARU95] could be used to link this library with the executable.

Since these computers do not have synchronized clocks, we cannot directly compare the writer's send time to the reader's receive time. Finally, if the writer writes the message long before the reader is ready to read the message, the message will be buffered on the reader's machine. This makes it difficult to estimate the time of reception. We will discuss these problems in some depth after summarizing the corresponding problems associated with estimating throughput.

In estimating throughput, we face similar challenges. Because we do not have control over the operating system, we have difficulty estimating transmission time. This affects our ability to estimate throughput. From an application's perspective, once it calls `read()`, it blocks and remains blocked until the operating system returns with data in the buffer. We could measure the total blocked time after an application makes a `read()` system call and assume that this elapsed time is an estimate of total transmission time. However, unless the `write()` that corresponds to the `read()` occurred at the same time, this assumption would most likely result in incorrect throughput estimates because of the composition of the blocked time.

We refer to two significant problems associated with estimating transmission time as the "early reader-late writer" problem and the "late reader-early writer" problem. The remainder of this section will further examine these problems.

In the "early reader-late writer" scenario, the reader calls `read()` and blocks waiting for the writer to execute `write()`. Some time after the writer finally writes, the reader receives the message and unblocks. In this case, the total blocked time is composed of both the time spent transmitting as well as the time spent waiting for the late writer. Because we cannot know how much of the blocked time was due to waiting for the late writer, we cannot assume that blocked time is equivalent to the transmission time. If we were to make such an assumption, we would underestimate throughput.

In the "late reader-early writer" scenario, the writer writes to the network on an established connection, but the reader has not yet called `read()`. The operating system on the reader's host may buffer some or all of the data received from the writer. When the reader finally calls `read()`, it reads the buffered portion of the message directly from local memory. In this case, the total blocked time is composed of time spent reading from local memory, as well as the time required to read the unbuffered portion of the message from the network. In most systems, retrieving data from memory is significantly faster than network transmission time, so using this total time would result in an overestimate of throughput.

In summary, unless the `read()` and `write()` calls happen at exactly the same time, we cannot simply use blocked time to estimate throughput. Additionally, we face problems related to clock offset in estimating latency. Our approach to measuring network QoS recognizes and takes advantage of the "early reader-late writer" scenario to aid in obtaining accurate latency and throughput estimates.

### 2. Observation That Helps

Many writes to the network by applications are large (e.g., files and graphics). As we saw in Netscape, a `read()` from the network returns immediately upon receiving data in the buffer. Even though an application writer specifies the amount to be read in the `read()` system call, the call will return with the amount of data actually read immediately upon receiving any data. To ensure that all desired data are read, the application writer must implement the application so that it repeatedly calls `read()` until it has read the entire message – this requirement is independent of whether an application writer wants to link his program with our client libraries. However, we will use this observation in conjunction with the "early reader-late writer" scenario to help estimate throughput.

### 3. Passive Network Monitoring Approach

In this section we describe the passive network monitoring approach that we developed for MSHN. We will give an overview of our approach and then address the following four areas: clock offset compensation, the cooperating writer, the cooperating reader, and special considerations.

#### *a. Overview of the MSHN Passive Network Monitoring Approach*

Our approach exploits the "early reader-late writer" scenario. We have wrapped the `read()` and `write()` library calls to recognize TCP connections. In our approach, the `read()` system call recognizes the "early reader-late writer" scenario, allowing the estimation of end-to-end latency, and when appropriate, throughput.

In measuring end-to-end latency, we must address the three problems raised above. The first is that we do not know when the writer sent its message. We solve this by wrapping the `write()` system call to append, to the front of the message, a timestamp from the writer's

clock. The second problem results from the fact that the reader's clock and the writer's clock are not synchronized, but the reader and writer need to have reference to a common timeline. We will address this problem in the next subsection. The final problem deals with the "late reader-early writer" scenario. We avoid this problem by detecting this situation and only calculating latency when we are sure that we are in the "early reader-late writer" scenario.

To estimate throughput, our mechanism must first estimate transmission time. In the "early reader-late writer" scenario, blocked time is composed of the end-to-end transmission time and the time spent waiting for the writer. We will exploit our observation that many network writes are large. As mentioned above, when a large message is read from the network, `read()` must be called repeatedly until the entire message is received. The two necessary components for calculating throughput are the number of bytes transmitted and transmission time for those bytes. Our algorithm computes the difference between the times of the first read and the last read. This difference is transmission time. Our technique "throws away" that first period of blocked time consisting of time due both to end-to-end data transmission as well as time spent waiting for the writer. It is safe to assume that the remaining time that it takes to read the message is due primarily to transmitting the remainder of the message. The number of bytes received will be the message size less the size returned by the first `read()`. This allows us to estimate throughput between the reader and writer.

We use the term "cooperative" to refer to the reading and writing applications that are linked with our library. We discuss cooperative readers and writers in more detail shortly.

### *b. Compensating for Clock Offsets*

Our passive network monitoring policy requires that the communicating hosts have access to a common time reference. Since we cannot assume that the member hosts of MSHN have perfectly synchronized clocks, we use a derivative of the Network Time Protocol (NTP) to estimate clock offsets between machines [COUL96].

In our modified version of this protocol, each communicating machine has a timeserver process running. The timeserver listens at an established port and communicates with a remote process, X, using the User Datagram Protocol (UDP). Upon receiving a query from process X, the timeserver gets a "timestamp" from its local clock and immediately writes it back to the process that makes the query.

The interaction between process X that wants to estimate its machine's clock offset with a remote computer, and the timeserver on that remote computer follows these steps:

1. Process X gets a local timestamp, $T_{i-2}$, from its host.
2. Process X sends a request for a timestamp to the timeserver on the remote machine.
3. Process X blocks waiting for the remote timeserver's reply.
4. The remote timeserver receives the request, gets a local timestamp, $T_{i-1}$, and sends this reply back to Process X.
5. Process X unblocks and gets another local timestamp, $T_i$.

The NTP assumes that the remote process recorded $T_{i-1}$ at the midpoint between $T_{i-2}$ and $T_i$. Process X can then estimate clock offset, $o_i$, as:

$$o_i = T_{i-1} - (T_{i-2} + T_i)/2$$

Error, $d_i$, can be estimated as being at least equal to half the round trip time:

$$d_i = (T_i - T_{i-2})/2$$

From the error formula, we can see that shorter round trip times yield smaller values for error. We exploit this observation by making multiple calls to the remote timeserver, and then keeping the offset that results from the shortest round trip time.

In MSHN, we would expect estimates of $o_i$ and $d_i$ to be available from the Resource Status Server (RSS), but prior to adding this functionality to the RSS, we implemented and tested our passive network performance monitoring algorithm by wrapping the `accept()` and `connect()` system calls to trigger these estimates. This extra code adds considerable overhead to these system calls. In the final implementation of MSHN the RSS would periodically, at times of low system usage, poll MSHN members and record clock offset and drift. To minimize added network traffic for delivery, the distribution of $o_i$ and $d_i$ can be included with security certificates [WRIG98].

### c. *The Cooperating Writer*

We incorporated into the MSHN library a wrapper for the `write()` system call that detects when `write()` has been called to write to a TCP connection. The wrapper appends the following information to the front of such messages: the writer machine's current time, $T_{remote'}$, and the size of the message.

### d. *The Cooperating Reader*

Like the `write()` system call's wrapper, the `read()` system call's wrapper also detects when it is reading from a TCP connection. Once it detects that the read is from a TCP connection, it then evaluates whether it is the first time that the `read()` is called for the particular data that are being received. We now enumerate the steps that the wrapper takes if it detects that this is the first time that the `read()` is being called for the particular data set:

1. Records a local clock time stamp, $T_{blocked}$, prior to (possible) blocking.
2. When the `read()` continues (unblocks), the augmented system call records the time, $T_{startRead}$.
3. $T_{remote'}$ and total message size is stripped from the first part of the message received.
4. $T_{remote'}$ is adjusted for clock offset between the reader and writer machines and this adjusted time is recorded as $T_{remote}$.
5. `read()` now tests to see whether the early reader, late writer situation has occurred. If $T_{blocked}$ occurred earlier than $T_{remote}$, then the reader was early. That is, the reader was blocked for a while waiting on the writer to write. Only in this case can end-to-end latency be approximated.

$$\text{Latency} = T_{startRead} - T_{remote}.$$

6. The received message data are passed to the application, with the return value of the `read()` system call decremented to account for the size of the timestamp and total message size fields.

We now describe the actions taken when the read wrapper is invoked after the initial `read()`.

1. The size of the message remaining is decremented by the amount of data in the buffer.

2. If the size of the message remaining is zero, the end of the message has been found. In this case, throughput can be calculated.

3. The `read()` wrapper calculates throughput using:

Throughput = (total message size – size of first part of message)/($T_{endRead} - T_{startRead}$)

We note that throughput is only calculated when more than one `read()` is invoked to obtain the data that was sent. We also note that the throughput and latency estimates are estimates of end-to-end throughput, which are, in fact, what our application is interested in.

### *e. Special Considerations*

There are two special considerations that must be taken into account when using this approach. First, latency and throughput can only be calculated for a subset of the total network traffic, that is, when the "early reader-late writer" scenario is true. We can modify the algorithm to increase the opportunities to estimate throughput by "loosening" the "early reader-late writer" requirement if we have a good estimate of the absolute minimum latency, $Latency_{min}$, between the communicating machines. Rather than requiring $T_{blocked}$ to occur before $T_{remote}$ (that is, $T_{blocked} - T_{remote} < 0$), we could allow $T_{blocked}$ to be up to the minimum latency later than $T_{remote}$ ($T_{blocked} - T_{remote} < Latency_{min}$). This seems insignificant for machines connected locally over high speed networks where latency is in the order of milliseconds or fractions of milliseconds. However, consider machines connected over extended network links, especially those using satellite communication. In this case, latency is in the order of 100's of milliseconds and this loosened requirement could prove significant. In this latter case, the opportunities to estimate throughput could increase dramatically with this modification.

The second consideration is that the approach that we presented is only intended for use with TCP network communication.

## C. SUMMARY

In our review of existing tools and application components, we classified the network monitoring techniques as either passive or active. Passive monitoring has the desirable attribute of minimal added overhead while active monitoring gives the ability to measure latency and is not tied to any particular application. We then presented an approach that makes use of the low overhead of passive monitoring, estimates end-to-end latency and throughput, and is independent of any particular application. Chapter VII will discuss observed overhead of this approach.

# VII. OBSERVED OVERHEAD

We recall from Chapter I that the three constraints that are imposed upon our method for gathering resource information are: (1) the implementation must not require any changes to the operating system; (2) modifications to the application code must be minimized; and (3) the overhead imposed by the information gathering mechanism should not be excessive. As described in Chapters IV and V, the use of the client library meets the requirements of the first two constraints. This chapter discusses the overhead of our client library. We first review our experimental setup for measuring this overhead, then describe each test in detail and present the results.

## A. EXPERIMENTAL SETUP

We identified three fundamental areas that we needed to measure: file I/O system calls, interprocess communication (IPC) system calls, and a "pure overhead" category. In the first two areas, we wanted to compare the performance of the wrapped system calls to that of the unwrapped system calls. In order to do this, we wrote a test program that made I/O and IPC related system calls. In order to measure unwrapped times, we simply compiled this program directly into executable code. For gathering the same information on the corresponding wrapped system calls, we compiled this program and linked it with our client library. In this way, we were able to measure the overhead imposed on the wrapped program by the I/O and IPC wrappers in our client library.

The "pure overhead" category consists of wrapped modules that could not be directly compared to unwrapped modules. The `exit()` system call provides an example of this. Because `exit()` does not return to the calling program, it is impossible to time the duration of the unwrapped `exit()` call. In these cases, we simply timed the duration of the functionality that we added and reported that number. In addition to the `exit()` wrapper, the `accept()` wrapper and the modified `MAIN()` function also fall into the "pure overhead" category.

We ran our non-network tests on an Intel Pentium Pro dual processor system. The system was equipped with 64 megabytes (MB) of random access memory (RAM), 512 kilobytes (KB) of cache memory, and two 166 megahertz (MHz) CPUs. The kernel of the Linux 2.0.29 operating system was compiled to support multiprocessor capabilities. For testing the remote

IPC calls, the Linux machine communicated over 10 megabit/second (Mb/s) Ethernet to a Sun Sparc 690 server with 64 MB of RAM and two SM100 Sparc processors running at 40 MHz. Finally, we used the department's network file server, which runs Sun's Network File Service (NFS), for testing overhead associated with accessing remote files.

For each test of file and IPC system call overhead, we made a large number of runs (100) for each wrapped and unwrapped test, alternating between the wrapped and unwrapped tests. We then took the mean of these 100 runs and compared the wrapped test times to the unwrapped times. For the "pure overhead" category, we took a similar approach. We modified the system call wrappers to output the total time added due to these wrappers, and then made 100 runs and calculated and reported the mean time for those calls.

In order to ensure that the time duration that we measured within each test exceeded the minimum precision of the clock, we repeated the system calls a sufficient number of times to obtain an elapsed time in excess of 0.05 seconds. In the following discussion, we will enumerate the number of iterations executed for each test. Some system calls could not be called a large number of times independently. An example of this is the `open()` system call for a file. If we open a large number of files, eventually we would exceed the number of file descriptors allocated per process. To overcome this, we paired the `open()` of each file with a corresponding `close()`. In our test, we opened a number of file descriptors and then closed them, repeating this sequence enough times to get a meaningful measure of duration. We took a similar approach for the `socket()`, `connect()`, `close()` sequence in the interprocess communication tests. We believe that this approach is justified as these system calls generally occur together; if an application opens a file, almost always, the application closes it. We set the optimization level to zero (-O0) in compiling the test program to avoid potential problems with compiler optimization of these repetitive test cases.

## B. EXPERIMENTS AND RESULTS

We now describe the experiments we performed and the results we obtained. We also provide plausible explanations for the few surprises that we encountered.

### 1. File Input and Output

We tested both local and remote file I/O. The local file I/O was to and from the Linux machine's local hard disk, while the remote file I/O was to and from the department's network file server. The results of each test were output to a log file.

#### *a. Local File Input and Output*

The steps taken by the local file tests are enumerated below:

- The test program for `open()` and `close()` opened and closed 20 local files a large number (10,000) of times. On the last iteration, it left the file open for the I/O tests that follow. The test program then calculated the total time elapsed and output this time to a log file.
- The test program executed 5,000 small (100 byte) `read()`'s from a local file. It then calculated the total time elapsed and output this time to a log file.
- The test program executed 50 large (50,000 byte) `read()`'s from a local file. It then calculated the total time elapsed and output this time to a log file.
- The test program executed 5,000 small (100 byte) `write()`'s to a local file. It then calculated the total time elapsed and output this time to a log file.
- The test program executed 50 large (50,000 byte) `write()`'s to a local file. It then calculated the total time elapsed and output this time to a log file.

The percentage overhead[8] observed is listed in Table 4. We did not find any real surprises in this data set. The `open()` and `close()` system calls showed relatively high overhead, which we expected. The opening and closing of a local file is extremely fast; we have added to this the entry of the resulting file descriptor into our `fdTable` data structure. This accounts for the 24.9% added overhead.

The `read()` and `write()` wrappers for file I/O add a constant amount of overhead regardless of the size of the I/O. Our observations are consistent with this fact. Small `read()` and small `write()` operations show a higher overhead than do the larger operations; the larger I/O operations take longer and so the relative overhead of the wrapper is small.

---

[8] Actual average run-times, from which these percentages were obtained, can be found in Appendix D.

| Compared System Call | Added Overhead of Wrapped System Call |
|---|---|
| `open()` and `close()` | 24.9% |
| Small `read()` | 9.5% |
| Large `read()` | 5.4% |
| Small `write()` | Less than 1% |
| Large `write()` | Less than 1% |

**Table 4: Local File I/O Wrapper Overhead**

*b. Remote File Input and Output*

The steps taken by the remote file tests are enumerated below:

- The test program for `open()` and `close()` opened and closed 20 remote files a large number (10,000) of times. On the last iteration, it left the file open for the I/O tests that follow. The test program then calculated the total time elapsed and output this time to a log file.
- The test program executed 5,000 small (100 byte) `read()`'s from a remote file. It then calculated the total time elapsed and output this time to a log file.
- The test program executed 50 large (50,000 byte) `read()`'s from a remote file. It then calculated the total time elapsed and output this time to a log file.
- The test program executed 250 small (100 byte) `write()`'s to a remote file. It then calculated the total time elapsed and output this time to a log file.
- The test program executed 50 large (50,000 byte) `write()`'s to a remote file. It then calculated the total time elapsed and output this time to a log file.

The overhead we observed is listed in Table 5. The most surprising result from this analysis is the relatively high overhead of the `open()` and `close()` operation. On further investigation we found that this overhead is due to our use of the `fstatfs()` system call, which we use to differentiate between local and remote files. While `fstatfs()` returns very quickly when querying about a local file, it takes considerably longer when querying about a remote file. It appears that `fstatfs()` must access the remote file server even when `open()` does not. We surmised that because this call is much rarer than remote `read()`'s and `write()`'s, it has not been optimized. It could easily be optimized in a distributed file system by caching a small amount of information locally.

| Compared System Call | Added Overhead of Wrapped System Call |
|---|---|
| `open()` and `close()` | 50.8% |
| small `read()` | 2.7% |
| large `read()` | Less than 1% |
| small `write()` | Less than 1% |
| large `write()` | Less than 1% |

**Table 5: Remote File I/O Wrapper Overhead**

Examining these I/O system calls, we see the effect of a longer system call on the relative overhead of our file I/O wrappers. Because `read()` and `write()` system calls generally take longer over a network file system, and the overhead added by our wrappers is relatively constant, the observed relative overhead is much lower than for the local file I/O.

Initially, we thought that we might be able to use the times required to `read()` from a remote file to estimate network throughput. However, since the actual network calls are executed at kernel level, the wrapper cannot distinguish when remote `read()` calls result in network traffic. Since part or all of a file being accessed may be cached on the local machine, due to an earlier `read()` or `write()` to this file, a `read()` from a remote file may actually be accomplished from local memory.

### 2. Interprocess Communication

We tested both local and remote interprocess communication. The local interprocess communication was tested by running a wrapped and an unwrapped server application on the Linux machine; the wrapped and unwrapped test programs on this machine act as clients and interact with wrapped and unwrapped local servers, respectively. We also ran the same server versions on the Sun machine in order to test remote interprocess communication. The client programs on the Linux machine acted as drivers; once again, we alternated between wrapped and unwrapped client applications. The results of each were then output to a log file that was later read to compare system call times.

#### *a. Local Interprocess Communication*

The steps taken by the local interprocess communication tests are enumerated below:

- The test program for `socket()`, `connect()` and `close()` opened, connected and closed a socket to a local process 100 times. On the last iteration,

the test program left the socket open and connected for subsequent tests. It then calculated the total time elapsed and output this time to a log file.

- The test program executed 5,000 small (100 byte) `read()`'s from a socket by a local process. It then calculated the total time elapsed and output this time to a log file.

- The test program executed 50 large (50,000 byte) `read()`'s from a socket by a local process. It then calculated the total time elapsed and output this time to a log file.

- The test program executed 5,000 small (100 byte) `write()`'s to a socket by a local process. It then calculated the total time elapsed and output to a log file.

- The test program executed 25 large (50,000 byte) `write()`'s to a socket by a local process. It then calculated the total time elapsed and output this time to a log file.

The overhead observed is shown in Table 6. The first entry in the table for the `socket()`, `connect()`, `close()` sequence surprised us. Our wrapped version consistently ran significantly faster than the unwrapped version of the program on the dual processor Linux machine, which was contrary to our expected results. Since our wrapper adds additional functionality, we should have seen that the wrapped version ran slower. We used the `strace()` utility to confirm that both the wrapped and unwrapped programs called the same system calls, which they did. We then tried to reproduce this phenomenon on a single processor Linux machine. We found that we could not reproduce these results on a single processor machine; the results of this run are the second entry in Table 6. Finally, we tried to reproduce these results on the Sun multiprocessor; the results of this run are the third entry in Table 6. Again, we were unable to cause this pattern to occur. We believe that the additional overhead of the wrapper may have caused the Linux multiprocessor kernel to schedule the process' context switching more optimally than the unwrapped version. This then resulted in a shorter runtime for the average wrapped run. In any case, wrapping this system call does not add significant overhead.

Looking at the second entry in Table 6, we see that the wrapped `socket()`, `connect()`, `close()` sequence on the single processor machine showed a relatively low overhead of 2.4%. Initially, we found this to be a bit strange as the `socket()` and

`connect()` wrappers have the same added functionality as the file `open()` wrapper, while the IPC `close()` and file `close()` wrappers have the same added functionality. Therefore, we expected to see an overhead percentage in the IPC wrapper similar to that of the file wrapper. On further inspection we found that the `socket()` and `close()` system calls took substantially less than a millisecond to complete, while the `connect()` system call took over a millisecond to complete. Because the functionality of these wrappers is so similar to the file I/O, we expect that the `socket()` and `close()` system calls add the same overhead as the local file `open()` and `close()` system calls, or approximately 25% of the local file I/O `open()` and `close()` time. Further, because the `connect()` wrapper has approximately the same functionality as the `open()`, it should add less than 25% of the local file I/O `open()` and `close()` time. Therefore the total overhead is substantially less than the total time to complete the wrapped `socket()`, `connect()`, `close()` sequence, resulting in a low percentage overhead.

| Compared System Call | Added Overhead of Wrapped System Call |
|---|---|
| `socket()`, `connect()`, `close()`[9] | – 27.0% |
| `socket()`, `connect()`, `close()`[10] | 2.4% |
| `socket()`, `connect()`, `close()`[11] | 5.7% |
| small `read()` | 6.5% |
| large `read()` | 1.5% |
| small `write()` | 2.1% |
| large `write()` | 1.8% |

**Table 6: Local IPC Wrapper Overhead**

[9] The first test ran on the Linux multiprocessor system described in section C, above.

[10] The second test ran on Linux single processor system with a 200MHz Intel Pentium processor, 128 MB of RAM, and 512KB of cache memory.

[11] The third test ran on the Sun multiprocessor system described in section C, above.

*b. Remote Interprocess Communication*

The steps taken by the remote IPC tests are enumerated below:

- The test program for `socket()`, `connect()` and `close()` opened, connected, and closed a socket to a remote process 100 times. On the last iteration, the test program left the socket open and connected for subsequent tests. It then calculated the total time elapsed and output this time to a log file.

- The test program executed 200 small (100 byte) `read()`'s from a socket by a remote process. It then calculated the total time elapsed and output this time to a log file.

- The test program executed 25 large (50,000 byte) `read()`'s from a socket by a remote process. It then calculated the total time elapsed and output this time to a log file.

- The test program executed 100 small (100 byte) `write()`'s to a socket by a remote process. It then calculated the total time elapsed and output this time to a log file.

- The test program executed 25 small (100 byte) `write()`'s to a socket by a remote process. It then calculated the total time elapsed and output to a log file.

The overhead we observed is shown in Table 7. While extremely high, the overhead that we see in the `socket()`, `connect()`, `close()` sequence is expected. We recall from Chapter VI that we modified both the `connect()` and `accept()` system calls to estimate clock offset with the remote host. As we discussed in Chapter VI, this modification requires that the local machine, executing the `connect()` or `accept()` system call, send a request and receive a response from a server running on the remote machine in order to estimate offset. Additionally, to ensure accuracy, this is repeated a number of times. Each invocation of this method requires approximately two times the average latency of the communication channel between the two machines, as well as some computational overhead. In the wrapper used in our experiment, `connect()` invoked this offset calculation method five times. This resulted in the observed overhead. We note that, as discussed in Chapter VI, the method that we used to estimate clock offset is somewhat artificial. We expect that, in the complete MSHN architecture, the RSS will calculate the drift and offset at times of small network load. The RSS will then distribute the relative clock offset and drift to the machines in conjunction with certificate distribution. Therefore, in the final version of MSHN, the overhead of the `accept()` and

connect() system calls will be reduced to approximately that of the corresponding local IPC calls.

| Compared System Call | Added Overhead of Wrapped System Call |
|---|---|
| socket(), connect(), close() | 705.8% |
| small read() | 69.7% |
| large read() | Less than 1% |
| small write() | 91.3% |
| large write() | 25.6% |

**Table 7: Remote IPC Wrapper Overhead**

We see relatively high overhead in our system call wrappers that implement our approach for calculating network QoS. Before further examining this, we feel compelled to note a few points about this overhead. While our approach does effectively slow down the perceived speed of system call execution, it adds little to the load of the network itself. This was one of the most important points of the passive versus active methods for monitoring network QoS. Additionally, although we see the overhead as a large percentage of the original time to complete the system call, the time to execute the system call itself is generally extremely short. For example, in our small read() example, the 69.7% added overhead that our wrapper added is equivalent to an additional 0.00027 seconds to each read().

Because, generally, write()'s do not block, they execute much faster than the corresponding read(). Therefore, any overhead that we add to the write() will show up as a more dramatic increase in relative overhead. The small write() overhead provides a dramatic example: although it took each write() 91.3% longer to execute, this was equivalent to only an additional 0.000024 seconds per write() system call.

We found that increasing the total volume of read()'s and write()'s to the network caused the perceived overhead of the write() system call to increase dramatically. Figure 21 shows a comparison of the perceived overhead of the write() system call with 100 write()'s versus 1,000 write()'s. The message size in each test was 100 bytes. This increase in overhead results from the effects of the slower reader. Because the reading process is wrapped, it reads from its buffer at a slightly reduced rate. At some point, the local writing process' buffer becomes full and the writer must block. In the wrapped scenario, blocking

happens earlier because the wrapped reader is slower in emptying its buffer. Therefore, the wrapped writer spends more time blocked. Because small `write()`'s normally happen much faster than `read()`'s, this slow down of the writer by the reader has a dramatic effect on the perceived overhead of the write.

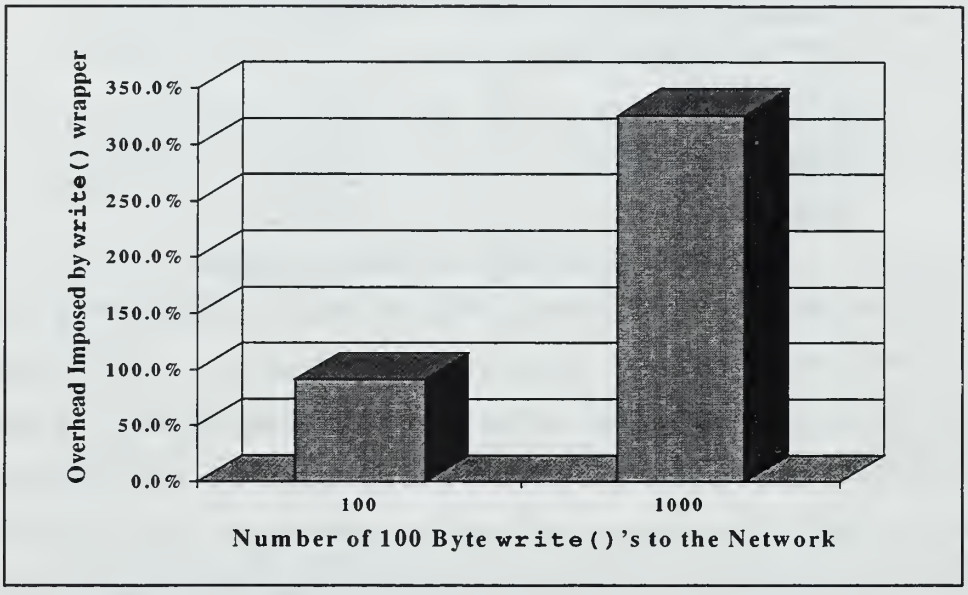


**Figure 21: Increasing Overhead of write() Wrapper with Increased Message Volume**

### 2. Overhead of Modules

As discussed previously, we could not directly compare every wrapped system call to the unwrapped version. The system calls that we could not directly measure are `exit()` and `accept()`. Additionally, we could not directly compare the overhead imposed by calling our method `MAIN()` before calling the original `main()`. So, the approach that we took was to make a number of runs and gather the actual added runtime for each of these calls. We ran these tests both on Linux multiprocessor machine and the Sun machine; the results are listed in Table 8 below.

The overhead associated with the `accept()` system call is comparable to that seen in the `connect()` system call. We note that `exit()` and `MAIN()` will only be called once per process, and so we can expect that our modification will add a net total of approximately 0.02 seconds to a wrapped process that runs on a machine of like architecture.

| Method | Added Overhead on multiproc Linux | Added Overhead on Sun server |
|---|---|---|
| `MAIN()` | $1 \times 10^{-5}$ seconds | $6 \times 10^{-5}$ seconds |
| `exit()` | 0.02 seconds | 0.02 seconds |
| `accept()` | 0.01 seconds | 0.02 seconds |

**Table 8: Overhead of `exit()`, `accept()` and `MAIN()`**

## C. SUMMARY

In summary, we can expect that, in most cases, the use of the wrappers in our client library will add a modest amount of overhead to an application. We found that the average percentage overhead incurred to the total run-time of our test program by the wrappers was a little less than 3%. Of course, we do not claim that our test program is representative of any application. Therefore, the overhead to a "real world" application may vary significantly from this percentage. This leads to the possibility of some future work in comparing the performance of established benchmark programs running wrapped and unwrapped.

# VIII. CONCLUSIONS AND FUTURE WORK

This thesis accomplished the three main objectives outlined in Chapter I. First, it describes, in extensive detail, a technique used to wrap Unix system calls; this description is supplemented by a system call wrapping tutorial in Appendix C. Second, it describes the implementation of a client library that uses the system call wrapping technique to monitor an application's resource usage as well as the load on the resources on which the application runs. Finally, it developed a new technique to passively monitor end-to-end network performance (throughput and latency). In this chapter we will discuss the contributions of this thesis and propose follow-on work.

## A. CONCLUSION

The Management System for Heterogeneous Networks (MSHN) requires the gathering of resource usage information from applications that run within the MSHN system and status information of the resources within the scope of the MSHN scheduler. The MSHN scheduler uses this information to make decisions. This thesis presented one method of gathering this information: the use of a client library.

Chapter IV of this thesis presents a technique used to wrap system calls. One of our goals in detailing this technique was to create a resource for future researchers. The approach that we used and presented was used by others for different purposes; as we mentioned previously, the Condor system was, to our knowledge, the first to implement system call wrappers. However, throughout our research, we were unable to find any documentation that we could use as a road map to make use of this approach. Instead, we spent quite a bit of time analyzing source code to reproduce this technique. It is our hope that our description, coupled with the tutorial presented in Appendix C, will provide a resource for future researchers wanting to replicate this technique.

We made extensive use of the wrapping technique in the implementation of the MSHN client library. By wrapping system calls, we were able to achieve our objective of monitoring resource usage and status, while requiring no modifications to the operating system or application source code. Additionally, we found that this technique added relatively low overhead to the wrapped application. While our approach requires that the client library be

linked to application object code, we propose that for follow-on research, the Executable Editing Library, developed at the University of Wisconsin-Madison, may provide a way to accomplish the same ends given only an executable file.

In developing ways to monitor the status of resources in the MSHN system, we developed a passive approach for measuring end-to-end quality of service across a network. We surveyed existing monitoring methods and found that passive monitoring has the desirable attribute of minimal added overhead while active monitoring gives the ability to measure latency and is not tied to any particular application. We then presented an approach that makes use of the low overhead of passive monitoring, estimates end-to-end latency and throughput, and is independent of any particular application. We found that, in most cases, the overhead associated with this technique is acceptable and is only evident as slightly increased processing time on the sending and receiving machines; this technique creates virtually no increase in the existing network load.

## B. FUTURE WORK

### 1. Dynamic Libraries

The method for wrapping system calls that we introduced in Chapter IV only applies to statically linked libraries. Dynamically linked, or shared, libraries are increasing in popularity; if our client library is unable to interact with dynamic libraries, users may choose not to link their application with the client library. In order for MSHN to be fully accepted, we need to consider interacting with dynamic libraries.

### 2. Port to Windows NT

The approach that we described for implementing the client library applies to a Unix environment. Desktop systems are increasing in popularity due to their affordability and rapidly increasing computational power. Indeed, the Department of the Navy has chosen Windows NT to be the operating system of choice in all future automation purchases [CINC97]. Although the purchases of systems running Unix and other operating systems are not proscribed, any purchaser who chooses one of these "non-standard" systems must be prepared to fully justify his decision. In recognizing this development, MSHN must be able to inter-operate in many

environments, including Windows NT. This means that we must be able to apply the client library to applications that run in a Windows NT environment.

### 3. Use Executable Editing Tools to Implement Wrapping

Our method requires access to object code. If object code is not available, as discussed more fully in Chapter IV, we may be able to use the Executable Editing Library (EEL) developed at the University of Wisconsin-Madison. Using these tools, we could modify the C library functions as they are defined in the executable code, effectively wrapping system call functions.

### 4. Develop Approach for Passively Measuring UDP Network QoS

The approach that we described in Chapter VI only applies to messages sent across the network using the Transmission Control Protocol. Because much network traffic uses the User Datagram Protocol (UDP), we need to develop a way to monitor network performance using this protocol.

### 5. Create a Wrapped Java Virtual Machine

Another approach to linking our client library with applications is to simply wrap a Java Virtual Machine (JVM). Whenever we desire to run an application written in Java on the MSHN system, we would not have to modify the application executable code. Rather, we would run these Java applications on a modified JVM. The client library, linked with this JVM, would update the RRD and RSS as the application ran on the modified JVM. This would accomplish two objectives. First, we would only have to link the client library with the JVM once; we would not have to make any modifications to the executable Java applications. Second, since "compiled" Java code is byte code, the wrapping technique that we described for linking with an application's object code would not work. Rather than having to develop a new technique, we could apply our proven approach to the JVM.

### 6. Use Benchmarks to Estimate Overhead of Wrapping an Application

In Chapter VII we measured and presented the overhead, per system call, associated with using our client library. However, this does not necessarily give a good indication of the

overhead that an application wrapped with our library will see. Applications do more than just make system calls, and the system calls that they make will vary in type, quantity and frequency based upon the application type. We concluded Chapter VII with our observation that the wrapped test program took, on average, a little less than 3% longer to run than the unwrapped test program. It would be worthwhile to conduct a similar test using an existing benchmark suite (e.g., the Numerical Aerospace Simulation (NAS) suite of benchmarks [NASA98]) in order to estimate the overhead to a wrapped program that is representative of "real world" applications.

## APPENDIX A: ACRONYMS

| | |
|---|---|
| ATO | Air Tasking Order |
| C3ISIM | Command, Control, Communication, and Intelligence Simulation |
| C4I | Command, Control, Communications, Computers, and Intelligence |
| CFG | Control-Flow Graph |
| Commserver | Communications Server |
| CPU | Central Processing Unit |
| DARPA | Defense Advanced Research Projects Agency |
| DoD | Department of Defense |
| EADSIM | Extended Air Defense Simulation |
| EEL | Executable Editing Library |
| ETC | Expected Time for Completion |
| fd | File descriptor |
| I/O | Input and/or Output |
| ICMP | Internet Control Message Protocol |
| IP | Internet Protocol |
| JTF ATD | Joint Task Force Advanced Technology Demonstrator |
| MB | Megabyte |
| Mb/s | megabit/sec |
| MHz | Megahertz |
| MSHN | Management System for Heterogeneous Networks |
| NCCOSC | Naval Command, Control, and Ocean Surveillance Center |
| NFS | Network File System |
| NFS | Network File Service |
| NTP | Network Time Protocol |
| NWS | Network Weather Service |
| QoS | Quality of Service |
| RDT&E | Research, Development, Testing, and Evaluation |
| RMS | Resource Management System |
| RRD | Resource Requirements Database |
| RSS | Resource Status Server |
| SA | Scheduling Advisor |
| TCP | Transmission Control Protocol |
| UDP | User Datagram Protocol |
| VHM | Virtual Heterogeneous Machine |

## APPENDIX B: MSHN LIBRARY ARCHITECTURE AND COMPONENTS

In this appendix, we describe the components that make up the MSHN client library and how we organize them into a cooperating entity.

### A. MSHN COMPONENTS

In the body of this thesis, we discussed mechanisms for gathering data. We now turn our attention to how we organize those mechanisms in MSHN. Our architecture exports two modules that the application writer must link with his code in order to run in the MSHN system:

- MSHN_syscall_lib.o, and
- the MSHN C Run-Time object file (MSHN_crt1.o or MSHN_crt0.o, depending upon the system and compiler).

As discussed in Chapter IV, the application writer must explicitly link with the MSHN C Run-Time object file in place of the system's C Run-Time object file. In addition, he must also link his application with MSHN_syscall_lib.o. This file contains the system call wrappers, the modified `MAIN()`, and the methods for monitoring resource usage and resource availability. Additionally, methods for updating the RSS and RRD are contained in this file. In the following paragraphs, we briefly discuss each of the files that go into making MSHN_syscall_lib.o (Figure 22).

MSHN_syscall_lib.cc contains the wrapper definitions and code for gathering resource usage information. This file also contains the instantiations for the MSHN_monitor_RRD_Class, the MSHN_export_RSS_Class and the hashClass objects. The first two of these objects provide methods for tracking and updating resource usage and availability information that will be delivered to the RRD and the RSS. The hashClass object is the data structure used to track file descriptors for input and output. Each of these objects will be discussed further below.

The library, libMSHNc.a, is the modified C library with redefined symbol names for the wrapped system call functions (to return to the example from the text of Chapter IV, this is where `READ()` is defined). Each of the wrapper functions, after pre-processing parameters, calls a function with a modified symbol name from this library to invoke the originally intended system call function.

MSHN_MAIN.cc defines `MAIN()`. `MAIN()` is called by MSHN_crt1.o. `MAIN()` records the start time of the process, saves the application name and input parameters, and calls `main()`.

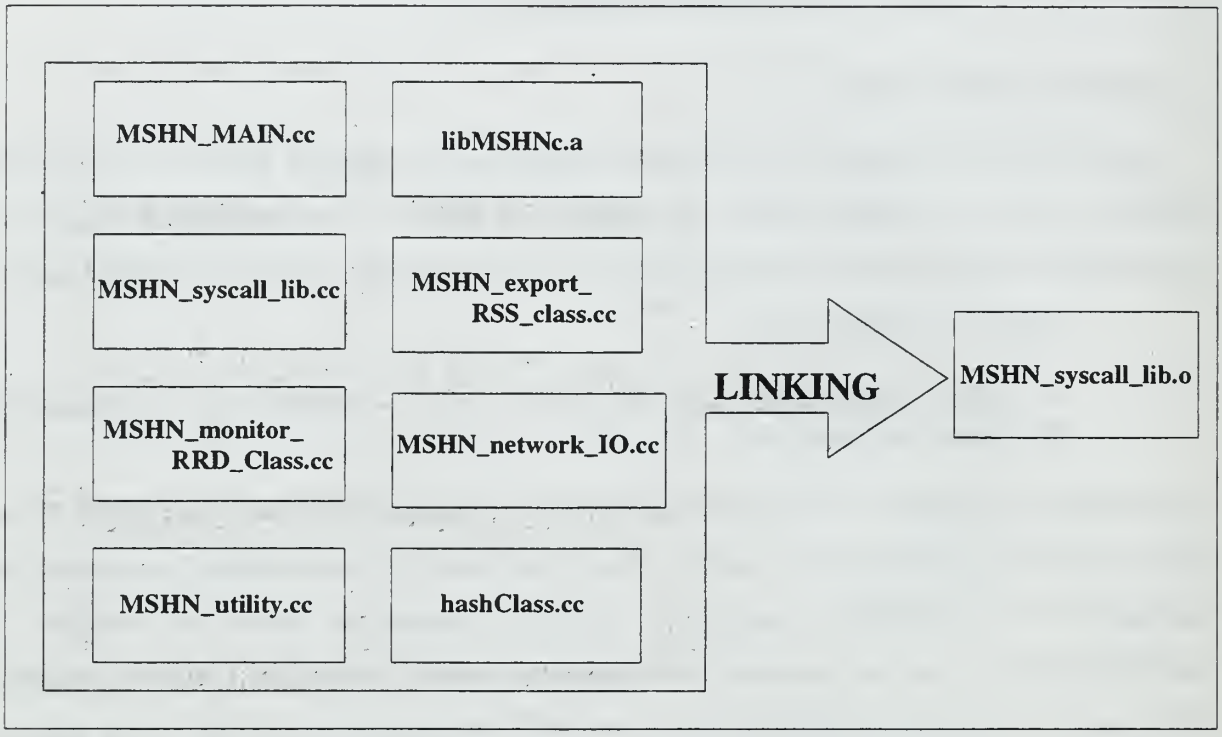


**Figure 22: Composition of MSHN_syscall_lib**

MSHN_export_RSS_Class.cc defines an object and methods used to send updates to the RSS. In Chapter VI, we presented an algorithm for calculating end-to-end throughput and latency. When these values become available, the methods in this class update the RSS (or its local proxy).

MSHN_monitor_RRD_Class.cc defines an object and methods used to track an application's resource usage. As the application runs, the wrappers in MSHN_syscall_lib update resource usage fields maintained by this object. Upon termination, this object sends an update to the RRD with the resources used by this application.

MSHN_network_IO.cc contains methods that implement the approach discussed in Chapter VI for passively calculating end-to-end throughput and latency across a network connection.

MSHN_utility.cc contains common methods used by all other files. MSHN_types.h defines types specific to MSHN resource monitoring.

The object defined in hashClass.cc contains a datastructure and methods to track open file descriptors and associated data. This data structure is crucial in differentiating different types of input and output, such as differentiating IPC from remote file access.

## B. INTERFACE TO SYSTEM

In Chapter III, we presented the MSHN architecture. In Figure 1, we diagrammed the architecture and showed the client library as a layer wrapped around the application. The application interacts with the library. The library communicates with the Resource Status Server (RSS), the Scheduling Advisor (SA), and the Resource Requirements Database (RRD). In addition, but not illustrated, is the implicit interaction of the application and client library with the machine's operating system and hardware. In Figure 23 below we expand the view of the client library.

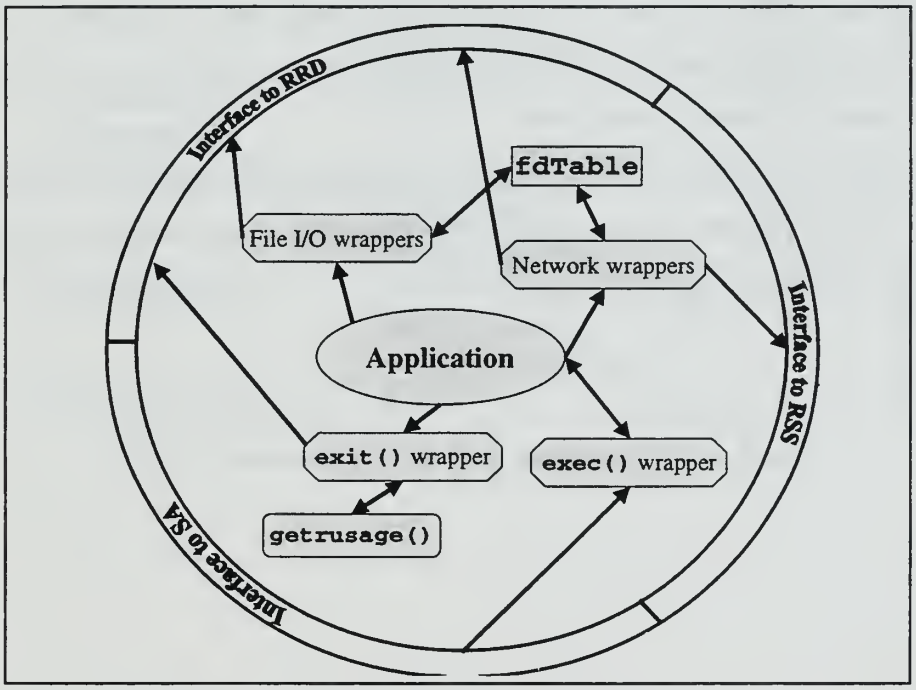


**Figure 23: Expanded View of Client Library**

Around the outer perimeter we show the interface to other portions of the MSHN architecture; again, these interface methods may interact directly with the actual MSHN component or to their local proxy. When the wrapped application begins running, it is authenticated via CDSA [WRIG98] and, along with this authentication, the CORBA ORB will

return references which the client will use to send RSS, RRD, and SA information [DUMA98]. The client library does not care, for example, whether it sends this information to the RSS, or to its local proxy.

We now provide an abbreviated description of some of the key methods used in the client library. As the application makes system calls, the calls are caught by the client library's system call function wrappers. File I/O wrappers record the size and type (local or remote) of file I/O. They also update totals stored in the RRD interface. Network I/O wrappers record the size of I/O, and they implement the passive network QoS measuring methods discussed in Chapter VI. These wrappers update the RRD and RSS interfaces. On application termination, the `exit()` wrapper calls `getrusage()` to obtain CPU usage information, and then invokes a method in the RRD to send all gathered resource usage information on the terminating process to the RRD. Also pictured is the interaction between the SA interface and an `exec()` system call function wrapper. This wrapper is not yet implemented, but is intended to show how the scheduling advisor might interact with the client library to start a process, or to cause the process to adapt to a higher or lower fidelity level.

In Chapter V, we presented a listing of the resources that the client library must monitor in Table 1. Table 9, below, extends Table 1 by listing all of the resources for which we have implemented a monitoring method.

| Resource | Metric | Implemented |
|---|---|---|
| Total Run-time | Seconds | Yes |
| CPU | Seconds | Yes |
| Memory | Maximum memory used | Yes |
| Cache Memory | Maximum cache used | No |
| Local disk | Bytes read | Yes |
| | Number of reads | Yes |
| | Bytes written | Yes |
| | Number of writes | Yes |
| Network disk | Bytes read | Yes |
| | Number of reads | Yes |
| | Bytes written | Yes |
| | Number of writes | Yes |
| Network | Bytes read | Yes |
| | Number of reads | Yes |
| | Bytes written | Yes |
| | Number of writes | Yes |
| Local inter-process communication | Bytes read | Yes |
| | Number of reads | Yes |
| | Bytes written | Yes |
| | Number of writes | Yes |
| Keyboard input | Number of bytes | Yes |
| | Seconds blocked waiting for user input | Yes |
| Power Consumption | Watts | No |

**Table 9: Resource Monitoring Implementation Status**

## APPENDIX C: TUTORIAL ON WRAPPING SYSTEM CALLS

This appendix provides a tutorial on how to wrap system calls, and how to catch the start of an application. The code shown was compiled and run on a Sun Sparc 690 server with 64 MB of RAM and two SM100 Sparc processors running at 40 MHz. This machine ran Sun OS version 4.1.3. In this tutorial, we will follow the same basic steps as outlined in Chapter IV, but we will focus on the "how-to" rather than on the higher level explanations from Chapter IV. A note on conventions: in order to save space and to facilitate a more logical flow, we will deviate from our previous technique of putting all code and screen output into figures. Instead, code and screen output will be indented and will be shown in `courier` font.

Our tutorial will follow the same steps as presented in Chapter IV:

1. Identify system calls to be wrapped.
2. Modify the C library's names of the system call functions that you are wrapping.
3. Write a wrapper function for each identified system call function. The wrapper will invoke the original system call function by using its new name.
4. Link the wrapper function with the modified C library into a composite library.
5. Link the application with the composite library.

Up to this point, we will have demonstrated how to wrap a system call. We will continue the tutorial by demonstrating how to wrap application start up, following the below listed steps:

6. Modify the C run-time object file.
7. Write a function, `MAIN()`, which will be invoked by the modified C run-time object file.
8. Link the compiled `MAIN()` with the composite library creating an augmented composite library.
9. Explicitly replace the linker's link with the system's C run-time object file with our modified C run-time object file; link the application with the composite library.

A key tool that we will use in wrapping the system calls is the `nm` user command. The `nm` command displays the symbol table of a file specified by the user; the Unix man pages provide a complete description of the use and functionality of this command [BERK91]. We

will use the nm command to see the symbols that exist within a file, and to see whether or not they are defined.

## A. IDENTIFY SYSTEM CALLS TO BE WRAPPED

In our simple example, we will wrap the `exit()` and `read()` system calls. The `read()` system call will be modified so that it counts the total number of bytes read, while the `exit()` system call will be modified so that, on application termination, it will output to the screen the number of bytes read during program execution, and the exit status of the application.

## B. MODIFY THE SYSTEM CALL FUNCTION NAMES IN THE C LIBRARY

We will make a modified copy of the C Library to provide a way to invoke the original system call function. The reasons for this step are described in detail in Chapter IV. The first thing that we will do is to copy the C library to a working directory.

```
hetero>     cp /usr/lib/libc.a ./
```

Next, we use the nm command on the C library so that we can locate where the `read()` and `exit()` symbols are defined. We search through the output of the nm command until we find the files in which these symbols are defined; the Unix man page tells us that a "U" indicates the symbol is not defined in that file, while a "T" indicates that it is defined.

```
hetero>     nm -a libc.a | more
```

Using this technique, we find that the `read()`, `exit()` and `_exit()` symbols are defined in files read.o, exit.o and _exit.o. (We discussed the difference between `exit()` and `_exit()` in Chapter IV, Section B). Since we want to modify these files, we extract them from the C library using the `ar` archive command. The `ar` command is used to create and modify archives, or libraries. Again, the Unix man pages [BERK91] give a more complete description of this command. We unarchive the desired files, and delete the C library from the working directory.

```
hetero>     ar x libc.a read.o exit.o _exit.o
hetero>     rm libc.a
```

Next, we use the nm command as described earlier to view the contents of each of these files. Note that the screen output is displayed immediately below the command prompt.

```
hetero>     nm -a _exit.o

00000000 T __exit
         U cerror

hetero>     nm -a exit.o

         U __cleanup
         U __exit
00000004 C __exit_handlers
00000000 T _exit

hetero>     nm -a read.o

00000000 T _read
         U cerror
00000018 t noerr
```

We see that, as indicated by the "T", _exit.o defines the `__exit` symbol, exit.o defines the `_exit` symbol, and read.o defines the `_read` symbol. Note that in this version of Unix, symbol names for functions generally are the function name with an underscore ("_") appended to the front.

The next step is to modify each of these symbol names. We use two simple programs, `makeUppercase` and `makeLowercase`, to accomplish this. The source code for these programs is listed in Appendix E. Each of these programs receives two command line parameters. The first parameter is the name of a file to open, the next is a sequence of characters to locate in that file. Once the sequence is found, it is raised to all uppercase or lowered to lowercase, depending on which program is run. Note that since we do not want to modify `__exit_handlers` in exit.o, we raise it to all uppercase and then bring it to all lower case after modifying `_exit`.

```
hetero>     makeUppercase _exit.o exit

hetero>     makeUppercase exit.o exit_handlers

hetero>     makeUppercase exit.o exit

hetero>     makeLowercase exit.o EXIT_HANDLERS

hetero>     makeUppercase read.o read
```

Next, we again use the nm command to see if we successfully modified the symbol names.

```
hetero>     nm -a _exit.o

00000000 T __EXIT
         U cerror

hetero>     nm -a exit.o

00000000 T _EXIT
         U __EXIT
         U __cleanup
00000004 C __exit_handlers

hetero>     nm -a read.o

00000000 T _READ
         U cerror
00000018 t noerr
```

We see that we successfully modified all of the symbol names. Note that in exit.o, we also modified the undefined reference to `__exit` to be `__EXIT`. This is correct. (First recall that in this version of Unix, all compiled symbol names have an underscore appended. So, `exit()` is compiled to the symbol `_exit`, and `_exit()` is compiled to the symbol `__exit`.) The original call, `exit()`, itself calls `_exit()`. Had we left `__EXIT` lowercase, then we would have invoked the wrapper that we wrote for the `_exit()` system call, effectively catching the same call twice. With our modifications, a call to `exit()` will result in `_EXIT` being referenced, which in turn will call `__EXIT` in our libmodc.a, thereby bypassing our wrapper of `_exit()`.

Next, we archive these object files into our own library, libModc.a. We then update the library's table of contents using the `ranlib` command [BERK91], and finally clean up by deleting the object files.

```
hetero>     ar rcv libModc.a _exit.o exit.o read.o

a - _exit.o
a - exit.o
a - read.o

hetero>     ranlib ./libModc.a

hetero>     rm *.o
```

We have now created a modified C library.

## C. WRITE WRAPPER FUNCTIONS

The next step is to declare our wrapper functions. The body of our header file, `wrapper.h`, follows. The use of `extern "C"{ ... }` instructs the compiler to compile the functions as C-style symbols. When we use the `nm` utility, we will see that the symbols `_read`, `_exit`, and `__exit` are defined. Had we not explicitly told the g++ (C++) compiler to compile these functions as C-style symbols, the symbols for `read()`, `exit()` and `_exit()` would have been "mangled." A mangled symbol name has parameter type and return type data appended to the symbol. This mangling of symbol names is what allows the overloading of functions in C++. However, the symbols in the C library are C-style, therefore our wrappers must also use this style.

```
//-------------------------------------------------------
//Title: wrapper.h
//Description: A small wrapper package that wraps the
//  read() and exit() and _exit() system calls
//-------------------------------------------------------
//
extern "C"{
   int read(int fd, char* buf, int len);

   void exit(int status);

   void _exit(int status);

}//end extern "C"
//end file wrapper.h
```

Next, we define the wrapper functions for `read()`, `exit()` and `_exit()`. Note that we define the prototypes of the functions that we modified and put into libModc.a. We use `extern` twice here, the first use again tells the compiler to compile the prototypes as C-style symbols. The second use of `extern` precedes each prototype declaration; these uses of `extern` tell the compiler that these function definitions will be provided at link time. The file, `wrapper.cc`, is displayed on the following page.

```
//-------------------------------------------------------------
//Title: wrapper.cc
//Description: A small wrapper package that wraps the read() and
//   exit() and _exit() system calls
//-------------------------------------------------------------
//
#include <iostream.h>
#include "wrapper.h"

//tell the compiler that we will provide these C-style functions
// at compile time
extern "C"{
   extern int READ(int fd, char* buf, int len);
   extern void EXIT(int status);
   extern void _EXIT(int status);
}//end extern "C"

static unsigned sizeOfReads = 0;

//-------------------------------------------------------------
//passes the read() system call on to the O/S, increments the
// sizeOfReads variable, and returns number of bytes read
//-------------------------------------------------------------
int read(int fd, char* buf, int len)
{
   int numBytesRead = READ(fd, buf, len);

   if(numBytesRead>0){
      sizeOfReads += numBytesRead;
   }//end if

   return (numBytesRead);
}//end read

//-------------------------------------------------------------
//outputs to the screen the total number of bytes read
// by the application, its exit status, and then exit()
//-------------------------------------------------------------
void exit(int status)
{
   cout<<"The application wrote "<<sizeOfReads<<" bytes"<<endl;
   cout<<"The exit code was "<<status<<endl;
   EXIT(status);
}//end exit()

//-------------------------------------------------------------
//outputs to the screen the total number of bytes read
// by the application, its exit status, and then _exit()
//-------------------------------------------------------------
void _exit(int status)
{
   cout<<"The application wrote "<<sizeOfReads<<" bytes"<<endl;
   cout<<"The exit code was "<<status<<endl;
   _EXIT(status);
}//end _exit()

//end file wrapper.cc
```

We compile the wrapper file. Since the wrapper will not be an executable, we use the -c option [BERK91].

```
hetero>     g++ -c wrapper.cc -o wrapper.o
```

Next, we use the nm command to view the `wrapper.o` object file. Note that the symbols _EXIT, _READ, and __EXIT are listed as undefined; the linker has not yet provided their definition. Also, note that _exit, _read, and __exit are defined.

```
hetero>     nm -a wrapper.o

         U _EXIT
         U _READ
         U __EXIT
00000000 t ___gnu_compiled_cplusplus
         U ___ls__7ostreamPCc
         U ___ls__7ostreamPFR7ostream_R7ostream
         U ___ls__7ostreamUi
         U ___ls__7ostreami
0000017c T __exit
         U _cout
         U _endl__FR7ostream
000000bc T _exit
00000000 T _read
00000240 d _sizeOfReads
00000000 t gcc2_compiled.
```

## D. LINK THE WRAPPER FUNCTION WITH THE MODIFIED C LIBRARY

The next step links our modified C library, libModc.a, with the compiled wrapper, wrapper.o, in order to defined the symbols _EXIT, _READ and __EXIT. The resulting wrapper is saved as completeWrapper.o.

```
hetero>     ld wrapper.o -L./ -lModc -o completeWrapper.o
```

We use the nm command to view completeWrapper.o. We see that _EXIT, _READ, and __EXIT are now defined.

```
hetero>     nm -a completeWrapper.o

00000268 T _EXIT
000002c8 T _READ
00002000 d __DYNAMIC
00000260 T __EXIT
00000020 t ___gnu_compiled_cplusplus
0000019c T __exit
```

```
00000004 C __exit_handlers
00002068 D _edata
00002068 B _end
00000838 T _etext
000000dc T _exit
00000260 t _exit.o
00000020 T _read
00002060 d _sizeOfReads
00000268 t exit.o
00000020 t gcc2_compiled.
000002e0 t noerr
000002c8 t read.o
00000020 t wrapper.o
```

We now have a completed wrapper library for `read()`, `exit()`, and `_exit()`.

## E. LINK THE APPLICATION WITH THE WRAPPER LIBRARY

To test our wrapper, we wrote a short application, main.cc. This application prompts the user to enter some characters, and to then hit return. The application then exits. The main.cc program is listed below.

```
//-------------------------------------------------------------
//Title: main.cc
//Description: A small driver program to test the read()
//   and exit() wrappers
//-------------------------------------------------------------
//
#include <iostream.h>


main(){

   char buf[100];

   cout<<"Enter something and hit return"<<endl;
   cout<<"> ";

   //the C++ library function cin uses read()
   cin>>buf;

   //exit() will be called implicitly at application termination
   return 0;

}//end main

//end file main.cc
```

In order to link this program with our wrapper, first we compile main.cc, and then we link the application with our wrapper library, completeWrapper.o, to create the executable, testProgram.

```
hetero>     g++ -c main.cc -o main.o

hetero>     g++ main.o completeWrapper.o -o testProgram
```

At the command prompt, we type the executable name, testProgram. We see that the wrapper intercepted the `exit()` system call, and output to the screen the total number of bytes read (demonstrating the successful wrap of the `read()` system call), and the exit status. This application was successfully wrapped.

```
hetero>     testProgram

Enter something and hit return
> Hello world!
The application wrote 13 bytes
The exit code was 0
```

## F. MODIFY THE C RUN-TIME OBJECT FILE

Next, we want to wrap application start up. We do this by modifying the C run-time object file. On the Sun machine that we used, the C run-time object file was crt0.o. We copy that file to a local directory, and rename it Mod_crt0.o. We use the nm command to view the symbol table, and we see a reference to _main.

```
hetero> cp /usr/lib/crt0.o ./Mod_crt0.o

hetero> nm -g Mod_crt0.o

         U __DYNAMIC
         U __exit
00000270 D _environ
         U _exit
         U _main
00000000 T start
         U start_float

We will change the symbol name, _main, to reference _MAIN.

hetero> makeUppercase Mod_crt0.o main
```

We again use the nm command to view the symbol table of the object file, and we see that we successfully modified the symbol name.

```
hetero> nm -g Mod_crt0.o

         U _MAIN
         U __DYNAMIC
```

```
         U __exit
00000270 D _environ
         U _exit
00000000 T start
         U start_float
```

## G. WRITE A FUNCTION, `MAIN()`, TO BE INVOKED BY THE MODIFIED C RUN-TIME OBJECT FILE

Now that we've modified the C Run-Time object file, we must provide a function `MAIN()`. Our function, `MAIN()`, is contained in file `MAIN.cc` as described below. The added functionality of `MAIN()` is that it saves the executable name, and outputs this to the screen when `main()` returns.

```
//-----------------------------------------------------------------
//Title: MAIN.cc
//Description: Stores the application executable name, calls
//  main(), and on main()'s return, outputs to the screen the
//  executable name.
//-----------------------------------------------------------------
//
#include <iostream.h>
#include <string.h>    //for strcat()

//tell the compiler that main() will be provided
// at link time
extern "C" {
   extern int main(int argc, char* argv[]);
}//end extern "C"

//declare this as a C function
extern "C" {
   int MAIN(int argc, char* argv[]);
}//end extern "C"


int MAIN(int argc, char *argv[]){

   const int EXE_NAME_SIZE = 25;

   char execName[EXE_NAME_SIZE];

   int returnVal;

   strcpy(execName, argv[0]);
   returnVal = main(argc,argv);
   cout<<"main() returned, executable name is "<<execName<<endl;
   return returnVal;

}//end MAIN

//end file MAIN.cc
```

## H. LINK THE COMPILED `MAIN()` WITH THE COMPOSITE LIBRARY

We will link the compiled MAIN.cc into the complete wrapper library. For simplicity, we will use a make file to accomplish this. The make file is listed below.

```
hetero> more wrapperMakefile


completeWrapper.o: wrapper.o MAIN.o libModc.a
        ld wrapper.o MAIN.o -L./ -lModc -o completeWrapper.o
        rm wrapper.o
        rm MAIN.o

wrapper.o: wrapper.cc
        g++ -c wrapper.cc -o wrapper.o

MAIN.o: MAIN.cc
        g++ -c MAIN.cc -o MAIN.o

hetero> make -f wrapperMakefile
```

## I. REPLACE THE SYSTEM'S C RUN-TIME OBJECT FILE WITH OUR MODIFIED C RUN-TIME OBJECT FILE

Next, we must have the linker link with our modified C Run-Time object file, Mod_crt0.o, rather than with the system's default C Run-Time object file. In order to do this, we take a two-step approach. First, we use the `-v` option for linking [BERK91] the wrapper, completeWrapper.o, with the application; using the `-v` option will cause the linker to output to the screen the complete link command. This is an interim step; we cannot use the resulting executable as it will not be linked with Mod_crt0.o. However, it will output the complete link command to the screen. The make file that we use to accomplish this first step is listed below.

```
hetero> more applicationMakefile

testProgram: main.o completeWrapper.o
        g++ -v main.o completeWrapper.o -o testProgram
        rm main.o

main.o: main.cc
        g++ -c main.cc -o main.o
```

The output from compiling and linking using applicationMakefile is shown below.

```
hetero> make -f applicationMakefile

g++ -c main.cc -o main.o
```

```
g++ -v main.o completeWrapper.o -o testProgram
 gcc -v main.o completeWrapper.o -o testProgram -lg++
Reading specs from /usr/local/lib/gcc-lib/sparc-sun-
sunos4.1/2.6.3/specs
gcc version 2.6.3
 /usr/local/lib/gcc-lib/sparc-sun-sunos4.1/2.6.3/ld -e start -dc -dp -o
testProgram /lib/crt0.o -L/usr/local/lib/gcc-lib/sparc-sun-
sunos4.1/2.6.3 -L/usr/local/lib main.o completeWrapper.o -lg++ -lgcc -
lc -lgcc
rm main.o
```

The second step is to create the actual make file that we will use to create the executable that is linked with our version of the C Run-Time object file. We use an editor to copy the full link command, listed above, to a new make file, applicationMakefile2, shown below. The full link command replaces the line that accomplished the linking in applicationMakefile, "g++ -v main.o completeWrapper.o –o testProgram". We modify the sequence (in the full link command), "`/lib/crt0.o`", to cause the linker to use our C Run-Time file, "`./Mod_crt0.o`". The modified make file is shown below, with the modifications in bold type.

```
hetero> more applicationMakefile2

testProgram: main.o completeWrapper.o
        /usr/local/lib/gcc-lib/sparc-sun-sunos4.1/2.6.3/ld -e start -dc
-dp -o testProgram ./Mod_crt0.o -L/usr/local/lib/gcc-lib/sparc-sun-
sunos4.1/2.6.3 -L/usr/local/lib main.o completeWrapper.o -lg++ -lgcc -
lc -lgcc
        rm main.o

main.o: main.cc
        g++ -c main.cc -o main.o
```

## J. LINK THE APPLICATION WITH THE COMPOSITE LIBRARY

Now we simply compile using the newly created make file and then run our test program. You can see from the output that we successfully caught the application start up using this procedure.

```
hetero> make -f applicationMakefile2

hetero> testProgram


Enter something and hit return
> Go Army, Beat Navy!
main() returned, executable name is testProgram
```

```
The application wrote 20 bytes
The exit code was 0
```

## K. SUMMARY

This appendix served as a tutorial for writing system call wrappers, and for catching the startup of an application. The tutorial used Sun OS 4.1.3. Note that your system may vary in name, location and composition of the C library; thus you may have to unarchive the C library in a working directory and use the nm command to hunt for the correct object files. This author strongly recommends writing small test programs as you progress through the process of creating wrappers – once you have written a system call wrapper for one system call, test it.

## APPENDIX D: TEST RESULTS

Chapter VII outlines the test procedures that we used to determine the overhead added by our application; we do not repeat our coverage of test procedures in this appendix. Rather, we give the "raw" numbers that we used to calculate percentage overhead in Chapter VII. This appendix follows the structure of Chapter VII; we list the results of local file I/O, remote file I/O, local IPC, remote IPC, total run-time, and "raw overhead" tests.

| Local File I/O Test Results | | | | | |
|---|---|---|---|---|---|
| test | open(), close() local file | read() sm loc file | read() big loc file | write() sm loc file | write() big loc file |
| Unwrapped (secs) | 0.26500514 | 0.04377777 | 0.094421 | 0.12987662 | 0.27076467 |
| Wrapped (secs) | 0.33104186 | 0.04795684 | 0.09952284 | 0.13080131 | 0.270769 |
| %overhead | 24.91903365 | 9.54610068 | 5.40328952 | 0.71197572 | 0.00159917 |

**Table 10: Raw Data for Local File Tests**

| Remote File I/O Test Results | | | | | |
|---|---|---|---|---|---|
| test | open(), close() remote file | read() sm rem file | read() big rem file | write() sm rem file | write() big rem file |
| Unwrapped (secs) | 0.00984138 | 2.8628597 | 14.094478 | 9.1133731 | 9.864908 |
| Wrapped (secs) | 0.01484136 | 2.9404091 | 14.104861 | 9.11493022 | 9.8715287 |
| %overhead | 50.80567969 | 2.70880896 | 0.07366715 | 0.01708613 | 0.06711365 |

**Table 11: Raw Data for Remote File Tests**

| Local IPC Test Results | | | | | | | |
|---|---|---|---|---|---|---|---|
| test | read() sm loc ipc | read() big loc ipc | write() sm loc ipc | write() big loc ipc | s,c,c [12] loc socket | s,c,c [13] loc socket | s,c,c [14] loc socket |
| unwrapped (secs) | 0.19210436 | 0.134939 | 0.18642837 | 0.13978866 | 0.94615758 | 0.1430201 | 1.012384 |
| wrapped (secs) | 0.20459111 | 0.13690715 | 0.19040276 | 0.14224203 | 0.69089037 | 0.1464194 | 1.070394 |
| %overhead | 6.499982614 | 1.45854794 | 2.13185901 | 1.75505653 | -26.979355 | 2.3767985 | 5.729998 |

**Table 12: Raw Data for Local IPC Tests**

---

[12] "s, c, c" indicates the sequence, `socket()`, `connect()`, `close()`. This first test was done on the Linux multiprocessor machine described in Chapter VII, Experimental Setup.

[13] This `socket()`, `connect()`, `close()` test was done on the Linux single processor machine described in Chapter VII, Experiments and Results, Local Interprocess Communication.

[14] This `socket()`, `connect()`, `close()` test was done on the Sun multiprocessor machine described in Chapter VII, Experimental Setup.

| Remote IPC Test Results | | | | | |
|---|---|---|---|---|---|
| test | read() sm rem ipc | read() big rem ipc | write() sm rem ipc | write() big rem ipc | s,c,c rem socket |
| Unwrapped (secs) | 0.07600747 | 3.4896277 | 0.00264186 | 1.4440362 | 0.1256548 |
| Wrapped (secs) | 0.12901621 | 3.507469 | 0.00505256 | 1.8141398 | 1.01250408 |
| %overhead | 69.74148725 | 0.51126657 | 91.2501041 | 25.6298007 | 705.782252 |

**Table 13: Raw Data for Remote IPC Tests**

| Comparison of Program Run-Times | |
|---|---|
| test | total run-time |
| unwrapped (secs) | 43.6297115 |
| wrapped (secs) | 44.90629672 |
| %overhead | 2.925953842 |

**Table 14: Raw data for Total Program Run-time**

| Total Added Run-time of Wrappers | | |
|---|---|---|
| wrapper tested for overhead | added run-time on Linux mp | added run-time on Sun mp |
| MAIN() | 0.00001 sec | 0.00006 sec |
| exit() | 0.0210021 sec | 0.0198769 sec |
| accept() | 0.0108971 sec | 0.0189439 sec |

**Table 15: Raw Data for Pure Overhead Category**

## APPENDIX E: SOURCE CODE FOR MODIFYING SYMBOL NAMES

### A. SOURCE CODE FOR `makeUppercase()`

```
//-------------------------------------------------------------
//Title: makeUppercase.cc
//Author: CPT Matt Schnaidt, USA
//Description: Opens a file, searches for an array of characters,
//    if found, raises all characters to uppercase, and closes
//    file.  If the array occurs more than once, all instances
//    of the array are modified.
//Inputs: At the command prompt, the user must enter file name
//    and character array to modify, e.g., the command line
//      makeUppercase myFile.o word
//    will cause all instances of "word" in myFile.o to be
//    changed to "WORD".
//Compiler:  g++ for Unix, g++ for Linux
//-------------------------------------------------------------
#include <unistd.h>
#include <stdio.h>
#include <strings.h>
#include <iostream.h>
#include <ctype.h>

//prototype
int putToUpper(char* fileName, char* replaceArray);


//-------------------------------------------------------------
//Title: main()
//Description: Prompts user and calls putToUpper until all
// arrays have been found and modified or end of file is reached.
//-------------------------------------------------------------
main(int argc, char* argv[]){

   //check that we received correct number of input args
   if(argc != 3){
      fprintf(stderr, "usage: %s file replaceAry\n", argv[0]);
      _exit(1);
   }//end if

   //loop until we do not find any instances of replaceAry
   do{
   }while(putToUpper(argv[1], argv[2]));

   return 0;

}//end main
```

```
//-------------------------------------------------------------
//Title: putToUpper()
//Description: Opens the file, searches for the replaceArray,
// if found, raises to uppercase, and returns true.  Else,
// if replaceArray not found, returns false.
//-------------------------------------------------------------
//
int putToUpper(char* fileName, char* replaceArray)
{
   int replaceSize;
   int tempInt;
   int ix;
   int stepsSinceStart;
   int doneFlag = 0;
   int wrongString = 1;


   //get the size of the string to replace
   replaceSize = strlen(replaceArray);

   //open the input file
   FILE* in_file;
   if((in_file = fopen(fileName, "r+"))<=(FILE*)NULL){
      printf("Problem opening file specified: %s\n", fileName);
   }//end if

   //look through the entire file
   while(!feof(in_file)){
      tempInt=getc(in_file);
      //initialize our step counter; we've getc'ed 1 char
      stepsSinceStart = 1;

      //if we got the first character of the string we're looking
      // for, check the rest of the string
      if ((unsigned char)tempInt==replaceArray[0]){

         ix = 1;
         //while we are not at the end of file, and while we haven't
         // found the entire string yet, keep looking
         while((!feof(in_file))&&(ix<replaceSize)){
            tempInt=getc(in_file);
            //we just got another character, so increment the counter
            stepsSinceStart++;

            //if this character does not match the corresponding one
            // in our search string, set wrong string flag and break
            if((unsigned char)tempInt != replaceArray[ix]){
               wrongString = 1;
               break;
            }
            //otherwise, we're still matching the search string, so
            // leave the wrong string flag false
            else{
               wrongString = 0;
            }//end if

            ix++;
```

```
        }//end while

        //if we found the correct string ...
        if(!wrongString){
           //removed +1 from stepsSinceStart
           //seek back to the beginning of the matched string
           fseek(in_file, -(stepsSinceStart), SEEK_CUR);

           //loop thru the string and replace lowercase to uppercase
           for(int jx = 0; jx < replaceSize; jx++){
              putc((toupper(replaceArray[jx])), in_file);
           }//end for

           doneFlag = 1;
           break;
        }
        else{
           //otherwise, begin searching at the character position after
           // where we started this search
           //fsetpos(in_file, replacePos);
           fseek(in_file, -(stepsSinceStart-1), SEEK_CUR);
        }//end if

     }//end while

     if(doneFlag){
        break;
     }//end if
  }//end while

  fflush(in_file);

  fclose(in_file);

  return doneFlag;
}//end putToUpper
```

//end file makeUppercase.cc

## B. SOURCE CODE FOR `makeLowercase()`

```
//-------------------------------------------------------------
//Title: makeLowercase.cc
//Author: CPT Matt Schnaidt, USA
//Description: Opens a file, searches for an array of characters,
//    if found, raises all characters to lowercase, and closes
//    file.  If the array occurs more than once, all instances
//    of the array are modified.
//Inputs: At the command prompt, the user must enter file name
//    and character array to modify, e.g., the command line
//      makeLowercase myFile.o word
//    will cause all instances of "word" in myFile.o to be
//    changed to "WORD".
//Compiler:  g++ for Unix, g++ for Linux
//-------------------------------------------------------------
#include <unistd.h>
#include <stdio.h>
#include <strings.h>
#include <iostream.h>
#include <ctype.h>

//prototype
int putToLower(char* fileName, char* replaceArray);


//-------------------------------------------------------------
//Title: main()
//Description: Prompts user and calls putToLower until all
// arrays have been found and modified or end of file is reached.
//-------------------------------------------------------------
main(int argc, char* argv[]){

   //check that we received correct number of input args
   if(argc != 3){
      fprintf(stderr, "usage: %s file replaceAry\n", argv[0]);
      _exit(1);
   }//end if

   //loop until we do not find any instances of replaceAry
   do{
   }while(putToLower(argv[1], argv[2]));

   return 0;

}//end main


//-------------------------------------------------------------
//Title: putToLower()
//Description: Opens the file, searches for the replaceArray,
// if found, changes to lowercase, and returns true.  Else,
// if replaceArray not found, returns false.
//-------------------------------------------------------------
//
int putToLower(char* fileName, char* replaceArray)
```

```
{
   int replaceSize;
   int tempInt;
   int ix;
   int stepsSinceStart;
   int doneFlag = 0;
   int wrongString = 1;


   //get the size of the string to replace
   replaceSize = strlen(replaceArray);

   //open the input file
   FILE* in_file;
   if((in_file = fopen(fileName, "r+"))<=(FILE*)NULL){
      printf("Problem opening file specified: %s\n", fileName);
   }//end if

   //look through the entire file
   while(!feof(in_file)){
      tempInt=getc(in_file);
      //initialize our step counter; we've getc'ed 1 char
      stepsSinceStart = 1;

      //if we got the first character of the string we're looking
      // for, check the rest of the string
      if ((unsigned char)tempInt==replaceArray[0]){

         ix = 1;
         //while we are not at the end of file, and while we haven't
         // found the entire string yet, keep looking
         while((!feof(in_file))&&(ix<replaceSize)){
            tempInt=getc(in_file);
            //we just got another character, so increment the counter
            stepsSinceStart++;

            //if this character does not match the corresponding one
            // in our search string, set wrong string flag and break
            if((unsigned char)tempInt != replaceArray[ix]){
               wrongString = 1;
               break;
            }
            //otherwise, we're still matching the search string, so
            // leave the wrong string flag false
            else{
               wrongString = 0;
            }//end if

            ix++;
         }//end while

         //if we found the correct string ...
         if(!wrongString){
            //removed +1 from stepsSinceStart
            //seek back to the beginning of the matched string
            fseek(in_file, -(stepsSinceStart), SEEK_CUR);
```

```
            //loop thru the string and replace uppercase with lowercase
            for(int jx = 0; jx < replaceSize; jx++){
               putc((tolower(replaceArray[jx])), in_file);
            }//end for

            doneFlag = 1;
            break;
         }
         else{
            //otherwise, begin searching at the character position after
            // where we started this search
            //fsetpos(in_file, replacePos);
            fseek(in_file, -(stepsSinceStart-1), SEEK_CUR);
         }//end if

      }//end while

      if(doneFlag){
         break;
      }//end if
   }//end while

   fflush(in_file);

   fclose(in_file);

   return doneFlag;
}//end putToLower
```

//end file makeLowercase.cc

## LIST OF REFERENCES


[ARMS97] Robert Armstrong, *Investigation of Effect of Different Run-Time Distributions on SmartNet Performance,* Master's Thesis, Naval Postgraduate School, Monterey, California, September 1997.

[BECK97] M. Beck, H. Boehme, M. Dziadzka, U. Kunitz, R. Magnus, and D. Verworner, *Linux Kernel Internels,* Addison Wesley Longman, Menlo Park, California, 1997.

[BERK91] Berkeley Unix Distribution, *Unix Man Pages*, March 1991.

[BUSH97] Tom Bush, *SC-21 Roadshow*, http:\\sc21.crane.navy.mil, September 1997.

[CARF99] Paul Carff, *Analysis on Resource Usage Information Granularity Required for Optimal Scheduling*, Naval Postgraduate School, Monterey, California, expected March 1999.

[CASE96] Fred Case, Christopher Hines, and Steven Satchwell, *Analysis of Air Operations During Desert Shield / Desert Storm,* U.S. Air Force Studies and Analyses Agency, 1991.

[CINC97] Joint IT-21 Message CINCLANTFLT/CINCPACFLT, *Information Technology for the 21st Century*, Pearl Harbor, HI, Mar 97.

[COND96] Condor Team, *Condor Source Code,* University of Wisconsin-Madison, 1996.

[CORM97] Thomas Cormen, Charles Leiserson, and Ronald Rivest, *Introduction to Algorithms*, MIT Press, Cambridge, Massachusetts, 1997.

[COUL96] George Coulouris, Jean Dollimore, and Tim Kindberg, *Distributed Systems, Concepts and Designs,* 2d Edition, Addison-Wesley, New York, 1996.

[DUMA98] Alpay Duman, *Design, The Use and Run-time Overhead of CORBA in MSHN Project*, Master's Thesis, Naval Postgraduate School, Monterey, California, September 1998.

[FISC88] Charles Fischer and Richard LeBlanc, Jr., *Crafting a Compiler,* Benjamin/Cummings Publishing, Menlo Park, California, 1988.

[HAYE94] W. Hayes-Roth and L. Erman, *The Joint Task Force Architecture Specification (JTFAS)*, Teknowledge Federal Systems, Palo Alto, California, 1994.

[HENS97] Debbie Hensgen and John Falby, *Draft Proposal for Institute for Joint Warfare Analysis Research*, Monterey, California, 1997.

[HEWL96] Information Networks Division, Hewlett-Packard Company, *Netperf: A Network Performance Benchmark, Revision 2.1*, February 1996.

[INOU97] Jon Inouye, Shanwei Cen, Calton Pu, and Jonathan Walpole. "System Support for Mobile Multimedia Applications," *NOSSDAV*, St Louis, Missouri, 1997.

[JOIN95] Joint Chiefs of Staff, *Joint Vision 2010*, Washington, D.C., 1995.

[JOIN97] Joint Chiefs of Staff, *Joint Doctrine Encyclopedia*, Washington, D.C., September, 1997.

[KIDD96] Taylor Kidd, Debbie Hensgen, Richard Freund, and Lantz Moore, "SmartNet: A Scheduling Framework for Heterogeneous Computing," *ISPAN*, 1996.

[KIDD98] Taylor Kidd, Debra Hensgen, Richard Freund, Matt Kussow, and Mark Campbell. *Compute Characteristics: A Useful Characterization of Job Run-times*. In preparation for submission (1998).

[KRES97] John Kresho, *Quality Network Load Information Improves Performance of Adaptive Applications,* Master's Thesis, Naval Postgraduate School, Monterey, California, September 1997.

[KRES98] John Kresho, Debra Hensgen, Taylor Kidd, and Geoffrey Xie, *Determining the Accuracy Required in Resource Load Prediction to Successfully Support Application Agility*, EURO-PDS98, 1998.

[LARU95] James Larus and Eric Schnarr, *EEL: Machine-Independent Executable Editing*, SIGPLAN PLDI 95, 1995.

[LEEC98] Craig Lee, James Stepanek, B. Michel, Ian Foster, Carl Kesselman, Rober Lindell, Soonwook Hwang, Joseph Bannister, and Alain Roy, *Qualis: the Quality of Service Component for the Globus Metacomputing System*, IWQoS98, Napa, California, 1998.

[LITZ92] M. J. Litzkow and M. Solomon, *Supporting Checkpointing and Process Migration Outside the UNIX Kernel*, in Proceedings of the Winter Usenix Conference, San Francisco, CA 1992.

[LITZ97] Michael Litzkow, Todd Tannenbaum, Jim Basney, and Miron Livny, *Checkpoint and Migration of UNIX Processes in the Condor Distributed Processing System*, Technical Report #1346, University of Wisconsin-Madison, April 1997.

[LIVN95] Miron Livny, Michael Litzkow, Todd Tannenbaum, and Jim Basney, *Checkpoint and Migration of UNIX Processes in the Condor Distributed Processing System*, Dr Dobbs Journal, February 1995.

[NAVA96] Naval Command, Control, and Ocean Surveillance Center, Research, Development, Test and Evaluation Division, code 422, 53140 Gatchell Road, San Diego, CA 92152-7400. *SmartNet Scheduling Tool v2.6 Users Guide*, June 1996.

[NETS98] Netscape Communications Corporation, *Netscape Communicator Source Code*, March 1998.

[NOBL97] Brian Noble, M. Satyanarayanan, Dushyanth Narayanan, James Eric Tilton, Jason Flinn, and Kevin R. Walker, "Agile Application-Aware Adaptation for Mobility," *Proceedings of the 16th Symposium on Operating Systems*, 1997.

[PARA97] Paradyn Project, *Paradyn Parallel Performance Tools User's Guide,* Release 2.0, University of Wisconsin-Madison, 1997.

[PORT97] Wayne Porter, *Appraisal of "Analysis of Air Operations During Desert Shield / Desert Storm": an Article by Maj. F. T. Case,* Naval Postgraduate School, 1997.

[PRUY95] Jim Pruyne and Miron Livny, *Interfacing Condor and PVM to harness the cycles of workstation clusters*, University of Wisconsin-Madison, 1995.

[PUCA96A] Calton Pu, Tito Autrey, Andrew Black, Charles Consel, Crispin Cowan, Jon Inouye, Lakshmi Kethana, Jonathan Walpole, and Ke Zhang, *Optimistic Incremental Specialization: Streamlining a Commercial Operating System*, Oregon Graduate Institute, 1996.

[PUCA96B] Calton Pu, Tito Autrey, Jonathan Walpole, Crispin Cowan, and Charles Krasic, *Fast Concurrent Dynamic Linking for an Adaptive Operating System*, Oregon Graduate Institute, 1996.

[SCHN99] Matt Schnaidt, Debra Hensgen, John Falby, Taylor Kidd, and David St. John, *Passive, Domain-Independent, End-to-End Message Passing Performance Monitoring to Support Adaptive Applications in MSHN,* submitted to OSDI 99.

[SILB98] Abraham Silberschatz and Peter Bae Galvin, *Operating Systems Concepts, 5th Edition*, Addison-Wesley, Menlo Park, California, 1998.

[SPRI97] Neil Spring, *Network Weather Service for Mentat 3.0 User's Guide*, October 1997.

[STAL98] William Stallings, *Cryptography and Network Security, Principles and Practice,* 2nd Edition, Prentice Hall, Upper Saddle River, New Jersey, 1998.

[STEV92] W. Stevens, *Advanced Programming in the UNIX Environment*, Addison-Wesley, 1992.

[WOLS97] Rich Wolski, Neil Spring, and Christopher Peterson, *Implementing a Performance Forecasting System for Metacomputing: The Network Weather Service*, SC97 Technical Paper, 1997.

[WRIG98] Roger Wright, David Shifflett, and Cynthia Irvine, *Security for a Virtual Heterogeneous Machine*, to appear in the Proceedings of the Twelfth Computer Security Applications Conference, Scottsdale, AZ, December 1998.

## INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center.....2
   8725 John J. Kingman Road, Ste 0944
   Ft. Belvoir, Virginia 22060-6218

2. Dudley Knox Library .....2
   Naval Postgraduate School
   411 Dyer Rd.
   Monterey, California 93943-5101

3. Chairman, Code CS.....1
   Computer Science Department
   Naval Postgraduate School
   Monterey, CA 93940-5000

4. Dr Debra Hensgen.....10
   Computer Science Department, Code CS
   Naval Postgraduate School
   Monterey, California 93943-5100

5. John S. Falby.....1
   Computer Science Department, Code CS
   Naval Postgraduate School
   Monterey, California 93943-5100

6. CPT Matt Schnaidt.....2
   14160 Hayrake Hollow
   Chelsea, MI 48118

7. Dr. Edward Limoges.....1
   675 Wateredge Drive
   Ann Arbor, MI 48105

8. Dr. Gary Koob.....1
   DARPA / ITO
   3701 North Fairfax Drive
   Arlington, VA 22203-1714

9. Mr. Jim Manley.....1
   1810 Embarcadero Road
   Palo Alto, CA 94303